Case study: a search engine for one inbox (319K messages)

Key takeaways

1. ‘Fast enough to run on every call’ is a median claim, not the worst.
2. Search should be a pure read; make the index update its own command.
3. Docs the agent reads become gospel; keep CLAUDE.md honest.

Three accounts, one local full-text index

- Indexes the Thunderbird local mbox cache across all three accounts
- The first alarm was simple: only 15 emails arrived, but the update was taking a long time
- Built around an optimization that worked great in testing and DoS’d itself in production

The first mystery: only 15 emails arrived

The hard part was noticing what question to ask the agent

What I saw	Why it was suspicious	Question to ask
A query ran much longer than usual	15 new emails should not imply a full mailbox pass	What work did update actually do?
Load climbed after each query	each search seemed to create more update work	Is search silently writing?
SSD activity spiked	the tool was touching far more data than the new mail	Which mbox files were reparsed?

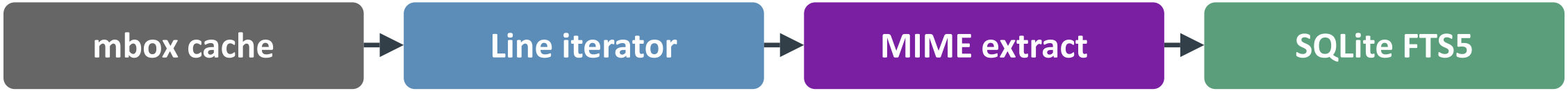
A good agent can help, but only after the human names the mismatch.

What email-index does

- Make every email searchable from the command line
  - Thunderbird stores all downloaded mail as ~90 mbox files totaling ~29 GB
  - email-search walks those files into a ~2 GB SQLite database with full-text search
  - 319,000 messages queryable in under 100 milliseconds with one command
- Serve an AI subagent, not a human reading inbox
  - The user has an ‘email’ subagent — when asked a question, it runs cross-account searches
  - ‘What did Hermes say about CXL?’ becomes ‘email-search from:hermes CXL’ in one shot
  - No clicking through Thunderbird folders; no IMAP latency
- Stay incremental — rebuilds are minutes, not hours
  - folder_state table caches (mtime, size, anchor offset) per folder
  - Common case — one new message appended to the tail — is one stat + one seek + one parse

Architecture — a pipeline with three sharp corners

Four stages turn ~29 GB of raw mbox into a searchable index — and three of them hide a sharp corner that only real mail triggers.



~90 files, ~29 GB	resilient, by key	charset + body decode	Porter stem, BM25
mtime lies: flag-rewrite vs. compaction	stdlib crashes on a non-ASCII From line	iso-8859-1 hides UTF-8 mojibake	external-content: index apart from text

By the numbers — code + index

Metric	Value
Python LOC	9,320 (4,257 src + 5,063 tests)
Tests	279 across 12 files
Commits	58 over ~2 months
Messages indexed	319,000 across 3 IMAP accounts
Source mbox size	~29 GB
Index DB size	~2 GB (~10× compression)

As of 2026-05. That ~10× compression is what makes full-text search across 319K messages feel instant.

By the numbers — tokens and cost

Metric	Value
Active days (ccusage)	6
Total tokens	38,500,000 (95% cache reads — heavy reuse)
Total cost (Claude Code)	$32.23
Most expensive day	2026-04-20 · $19.26 · bench-infra + locking refactor
Models used	Opus 4.6, Haiku 4.5

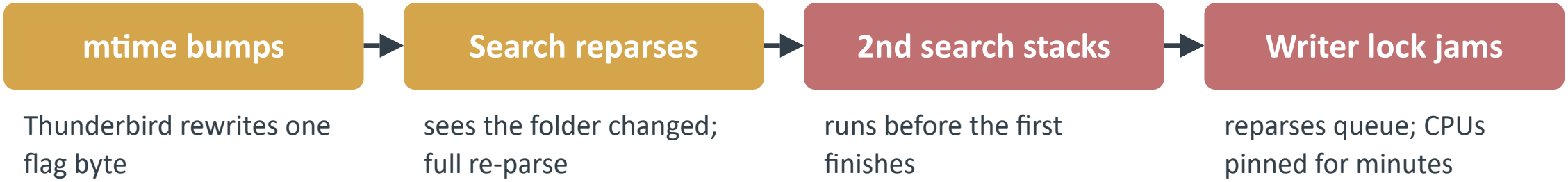
As of 2026-05. Heavily undercounted — ccusage data starts 2026-04-18, but the project began 2026-03-19. The initial-build day (17 commits) and the 2026-03-26 incremental-indexing day (15 commits) are not in these numbers.

The optimization that sounded smart

- The original design — search was the only command
  - User runs ‘email-search query’; the tool returns hits
  - If new mail arrived since the last run, the user got stale results
  - Awkward to remember ‘oh I should run an update first’
- The ‘obvious’ improvement (2026-03-26)
  - Add a silent incremental-update step BEFORE every search
  - Stat-check each folder; reparse only changed ones
  - Measured at 6 ms in testing — ‘fast enough to run silently before every search’
- Shipped. Worked great in testing. The user moved on.
  - For three weeks, no one noticed anything wrong
  - The cost of being wrong scales with how invisible the wrongness is

Three weeks later — the tool DoS'd itself

The optimization built to make queries faster made them dramatically slower.

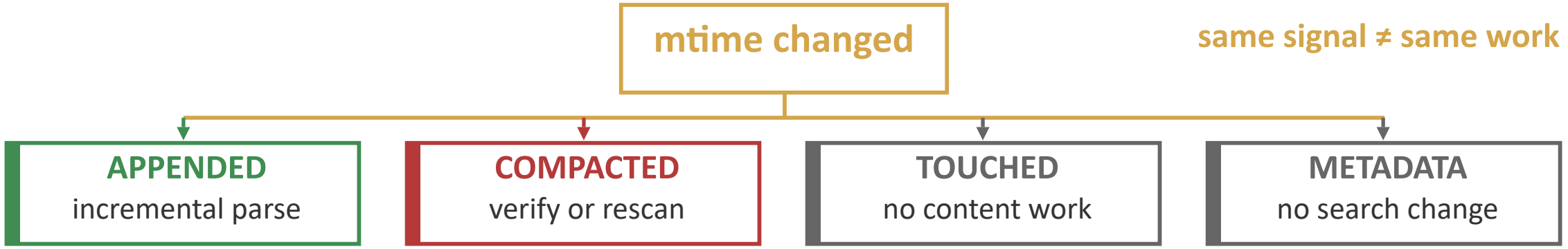


Lesson: 'fast enough to run on every call' is a claim about the median case, not the worst case.

The design keyed on the wrong signal

mtime tells you something changed; it does not tell you what work is needed

Signal	What it can tell you	What it cannot tell you
mtime	the file was touched	whether content relevant to search changed
size	bytes were appended or removed	which messages are new or gone
Message-ID set	which logical messages exist	whether you can skip parsing all MIME
anchor offset + message id	whether the prefix still matches	whether compaction changed the file



The fix — separate read from write

- Search becomes pure-read — no lock contention, no surprises in concurrent use
- Update is now an explicit subcommand the user (or a cron job) calls when convenient
- A separate cheap staleness check answers ‘is the index stale?’ without ever locking

Before — search silently updates

```
def search(query):
    update()  # 6 ms stat-check
    return fts5(query)

# concurrent searches all contend
# on the writer lock when ANY
# folder's mtime moves
```

After — read and write are different commands

```
def search(query):   # pure read
    return fts5(query)

def update():        # explicit
    for f in changed_folders():
        with write_lock():
            reindex(f)

def info(check=True):  # pollable
    return all_fresh()
```

source: HISTORY.md @ 2026-04-15; commit 709de19

Story — the docs were wrong about FTS5

- CLAUDE.md claimed FTS5 external-content deletes were impossible
- So the agent rebuilt the whole index instead of deleting rows
- Design pressure forced a recheck — the doc was wrong (2424b18)

Before — ‘impossible’, so rebuild

```
# CLAUDE.md said external-content
# FTS5 deletes are impossible, so:
def reindex(folder):
    drop_index()    # nuke it all
    rebuild()       # full re-parse
```

After — INSERT-delete with old values

```
# external-content delete = a special
# INSERT carrying the OLD values:
fts.execute(
    "INSERT INTO fts(fts, rowid, subject,"
    " body) VALUES ('delete', ?, ?, ?)",
    (rowid, old_subject, old_body))
# then remove the real row:
db.execute("DELETE FROM messages ...")
```

source: email-index HISTORY.md; commit 2424b18

Story — the spell checker flagged real words

- Goal: suggest ‘memory’ when the user types ‘memmory’
- fts5vocab lists every indexed token — the obvious source
- But it returns Porter stems, so correct words look misspelled

Before — vocabulary from fts5vocab

```
# fts5vocab yields Porter stems:
#   memory   -> memori
#   computer -> comput
for term in fts5vocab(conn):
    vocab.add(term)    # a stem!
```

After — tokenize the raw message text

```
rows = conn.execute(
  "SELECT subject, body FROM messages")
for subj, body in rows:
  for w in TOKEN_RE.findall(
      (subj + body).lower()):
    vocab[w] += 1   # surface form
```

source: email-index src/email_index/spelling.py

Approximate search is a product decision

Typo tolerance costs memory and latency; it should not slow correct queries

Search path	Cost	User value
raw query	~10 ms	fast path for normal searches
edit distance 1	~1 s from 15 MB cache	catches one missing, swapped, or wrong character
edit distance 2	~4 s from 34 MB cache	catches rougher misspellings, but feels expensive
build dictionary every time	~25 s	not acceptable on a search path

Run expensive correction only after the exact query returns nothing.

The index size is a policy knob

More tolerance means more precomputed structure

Keep

Vocabulary terms seen at least three times. That drops one-off URLs and accidental typos from the dictionary.

Set the frequency floor before you size the index.

Cache

Precompute the SymSpell delete structures for edit distance 1 and 2, and store them on disk.

Build the cache offline, never per query.

Delay

Do not load the cache at all until an exact query has already returned nothing to show.

Pay for tolerance only on a failed search.

Typo tolerance is a storage decision: choose the edit distance you are willing to store, then keep it off the fast path.

Story — lock byte 0x7FFF0000 on Windows

- Unix has flock; Windows has msvcrt.locking — must lock something
- Convention (portalocker, SQLite): lock one byte at 0x7FFF0000
- The first cut read every OSError as ‘another build is running’

Before — every error reads as contention

```
try:
    msvcrt.locking(fd, LK_NBLCK, 1)
except OSError:
    # ANY error looks like a lock:
    raise UpdateLockHeld
```

After — only EACCES / EDEADLK

```
os.lseek(fd, 0x7FFF0000, SEEK_SET)
try:
    msvcrt.locking(fd, LK_NBLCK, 1)
except OSError as e:
    if e.errno in (EACCES, EDEADLK):
        raise BlockingIOError
    raise    # a real error
```

source: email-index src/email_index/locking.py

Feature creep — the inverse pattern

- When designs got muddled, the agent DELETED code
  - Auto-update before every search — not patched, deleted: ‘no reasonable locking fix; the right fix is to not do it at all’
  - build/update overlap — deleted: ‘build always means full rebuild now; the overlap is gone’
  - Maintenance refactor in May — explicitly removed internal compatibility paths instead of preserving every historical import
- When new features landed, they shipped DELIBERATELY NARROW
  - email-refile (2026-05-07) — same-account moves only; no cross-account copy/delete; no fallback when the server lacks IMAP MOVE
  - HISTORY uses the phrase ‘deliberately narrow’ as a virtue, not a limitation
- The lesson — agent-built code can be disciplined if the agent is told to delete instead of accumulate
  - Less accumulation than a typical hand-built tool of this scale
  - Deletion as a primary design move, not a cleanup afterthought

Takeaways

1. ‘Fast enough to run on every call’ is a median-case claim — the worst case under contention is a different story
2. Read/write boundaries belong in separate commands, not branches in one command — surprises live at the seam
3. Docs the agent reads first become gospel until something forces a recheck — keep CLAUDE.md honest