Step1: Data Processing

```
import json
from pathlib import Path
from typing import List, Tuple, Dict, Any

import fitz  # PyMuPDF
import pandas as pd

# Project paths
ROOT = Path("/content/drive/MyDrive/RAG-FT-DATA")
RAW_DIR   = ROOT / "raw"
CLEAN_DIR = ROOT / "cleaned_text"
PROC_DIR  = ROOT / "processed"

CLEAN_DIR.mkdir(parents=True, exist_ok=True)
PROC_DIR.mkdir(parents=True, exist_ok=True)

list(RAW_DIR.glob("*.pdf"))
```

```
[PosixPath('/content/drive/MyDrive/RAG-FT-DATA/raw/annual-report-2024.pdf'),
 PosixPath('/content/drive/MyDrive/RAG-FT-DATA/raw/infosys-ar-25.pdf')]
```

```
!pip install PyMuPDF
```

```
Collecting PyMuPDF
  Downloading pymupdf-1.26.3-cp39-abi3-manylinux_2_28_x86_64.whl.metadata (3.4 kB)
Downloading pymupdf-1.26.3-cp39-abi3-manylinux_2_28_x86_64.whl (24.1 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 24.1/24.1 MB 65.2 MB/s eta 0:00:00
Installing collected packages: PyMuPDF
Successfully installed PyMuPDF-1.26.3
```

Utility Helpers

```
import re
import unicodedata
from typing import List, Iterable, Optional

# Precompile common patterns (extend as needed)
PAGE_NO_RE          = re.compile(r"^(page\s*)?\d{1,4}\s*(of\s*\d{1,4})?$", re.I)
YEAR_STANDALONE_RE  = re.compile(r"^(19|20)\d{2}$")
RUNNING_HEAD_HINTS  = [
    r".*\bannual\s+report\b.*",
    r".*\bintegrated\s+report\b.*",
    r".*\b(consolidated|standalone)\s+financial\s+statements\b.*",
    r".*\b(management\s+discussion\s+and\s+analysis|md&a)\b.*",
    r".*\bboard\s+report\b.*",
    r".*\bcorporate\s+governance\b.*",
]
RUNNING_HEAD_RES = [re.compile(p, re.I) for p in RUNNING_HEAD_HINTS]

# Lines that are usually noise
DROP_EXACT_RES = [
    PAGE_NO_RE,
    YEAR_STANDALONE_RE,
    re.compile(r"^confidential$", re.I),
    re.compile(r"^forward[- ]looking\s+statements$", re.I),
]

# Lines that *contain* these tokens are likely boilerplate headers/footers
DROP_CONTAINS_RES = [
    re.compile(r"\b(infosys|tata|integrated|annual\s+report|report\s+\d{4}|registered\s+office)\b", re.I),
]

BULLET_PREFIX_RE   = re.compile(r"^\s*[\u2022\-\u2013\*]\s+")
MULTISPACE_RE      = re.compile(r"\s+")
NBSP_RE            = re.compile(r"\u00A0")
INSIDE_NUM_SPACE_RE= re.compile(r"(?<=\d)\s+(?=\d{3}\b)")  # join 4 189 → 4189 (simple)
TRAILING_DOT_LINE  = re.compile(r"^\.+$")                  # dotted leaders
```

```
TRAILING_DOT_LINE  = re.compile(r" \.+$")                # dotted leaders
ONLY_PUNCT_RE      = re.compile(r"^[\.\—–•\*]+$")

def _mostly_caps(s: str, min_len=8, ratio=0.7) -> bool:
    """Heuristic: running heads often ALL/mostly caps."""
    if len(s) < min_len:
        return False
    letters = [ch for ch in s if ch.isalpha()]
    if not letters:
        return False
    caps = sum(ch.isupper() for ch in letters)
    return caps / max(1, len(letters)) >= ratio

def _unicode_norm(s: str) -> str:
    # normalize weird Unicode (smart quotes, ligatures, NBSP)
    s = unicodedata.normalize("NFKC", s)
    s = NBSP_RE.sub(" ", s)
    return s

def _should_drop_line(s: str) -> bool:
    if not s:
        return True
    if ONLY_PUNCT_RE.match(s) or TRAILING_DOT_LINE.match(s):
        return True
    # exact-pattern drops
    for rx in DROP_EXACT_RES:
        if rx.fullmatch(s):
            return True
    # contains-pattern drops
    for rx in DROP_CONTAINS_RES:
        if rx.search(s):
            # short + boilerplate-ish → drop; long paragraphs we keep
            if len(s) < 80 or _mostly_caps(s, min_len=6, ratio=0.6):
                return True
    # generic short ALL-CAPS junk
    if len(s) <= 3 and s.isupper():
        return True
    # running head in mostly caps
    if  mostly_caps(s):
```

```
            for rx in RUNNING_HEAD_RES:
                if rx.search(s):
                    return True
        return False

    def _post_token_normalize(s: str) -> str:
        s = INSIDE_NUM_SPACE_RE.sub("", s)      # 4 189 → 4189 (conservative)
        s = MULTISPACE_RE.sub(" ", s).strip()
        return s

    def clean_lines(lines: List[str], keep_table_like: bool = True) -> List[str]:
        """
        Robust line-level cleanup:
          - Unicode normalize; remove NBSP
          - drop page headers/footers/running titles/standalone years/page nos
          - drop short ALL-CAPS junk and punctuation-only leaders
          - optional: keep table-ish lines (contain lots of separators/numbers)
          - fix line-end hyphenation across lines (finan- \n cial → financial)
          - normalize spaces; repair split thousands
        """
        # 1) unicode & basic strip
        lines = [_unicode_norm(ln).strip() for ln in lines if ln is not None]
        lines = [ln for ln in lines if ln]  # drop empties

        # 2) first pass: drop obvious junk
        filtered: List[str] = []
        for s in lines:
            if _should_drop_line(s):
                continue
            # allow table-ish lines if requested (they contain many digits/separators)
            if keep_table_like:
                sep_count = sum(s.count(ch) for ch in ["|", "·", "•", ":", ";", "\t"])
                digit_count = sum(ch.isdigit() for ch in s)
                if sep_count >= 2 or digit_count >= 8:
                    # keep as-is (but normalize spaces)
                    filtered.append(_post_token_normalize(s))
                    continue
            # remove bullet prefix (•, -, –)
```

```
            s = BULLET_PREFIX_RE.sub("", s)
            filtered.append(_post_token_normalize(s))

        if not filtered:
            return []

        # 3) fix hard line-break hyphenation across lines
        # join s[-1] that ends with hyphen with next line if looks like a split word
        dehyphenated: List[str] = []
        i = 0
        while i < len(filtered):
            cur = filtered[i]
            if cur.endswith("-") and (i + 1) < len(filtered):
                nxt = filtered[i + 1]
                # Join if next starts with lowercase/alpha (likely a broken word)
                if nxt and nxt[0].isalpha() and nxt[0].islower():
                    cur = cur[:-1] + nxt  # drop hyphen, concatenate
                    i += 2
                    # normalize any double spaces created
                    cur = MULTISPACE_RE.sub(" ", cur).strip()
                    dehyphenated.append(cur)
                    continue
            dehyphenated.append(cur)
            i += 1

        # 4) merge tiny orphan lines into following paragraph (heuristic)
        merged: List[str] = []
        for s in dehyphenated:
            if merged and len(s) < 40 and not s.endswith("."):
                # append to previous if previous is short-ish and not ended
                prev = merged.pop()
                combined = (prev + " " + s).strip()
                merged.append(MULTISPACE_RE.sub(" ", combined))
            else:
                merged.append(s)

        return merged
```

Metric Patterns & Extraction

```
import re, json, unicodedata
from pathlib import Path
from typing import Tuple, List, Dict, Any

# optional OCR deps
try:
    from PIL import Image
    import pytesseract
    HAS_OCR = True
except Exception:
    HAS_OCR = False

# --- helpers (you already have these, included here for completeness) ---
# clean_lines: the improved version you adopted earlier
# detect_section: uses SECTION_PATTERNS

def _norm_unicode(s: str) -> str:
    s = unicodedata.normalize("NFKC", s).replace("\u00A0", " ")
    s = re.sub(r"[ \t]+", " ", s)
    return s.strip()

def _extract_text_text_mode(page) -> str:
    return page.get_text("text") or ""

def _extract_text_blocks_mode(page) -> str:
    # Reassemble from blocks in visual order
    try:
        blocks = page.get_text("blocks") or []
        # sort by (y1, x1)
        blocks.sort(key=lambda b: (round(b[1], 1), round(b[0], 1)))
        parts = []
        for (x0, y0, x1, y1, txt, block_no, block_type) in blocks:
            if txt:
                parts.append(txt)
```

```
            return "\n".join(parts)
        except Exception:
            return ""

    def _extract_text_words_mode(page) -> str:
        # Reassemble from words → lines by y position
        try:
            words = page.get_text("words") or []
            if not words:
                return ""
            # sort by y, then x
            words.sort(key=lambda w: (round(w[3], 1), round(w[0], 1)))
            lines, current_y, buf = [], None, []
            for (x0, y0, x1, y1, word, block, line, wno) in words:
                y = round(y1, 1)
                if current_y is None:
                    current_y = y
                # new line if y jumps
                if abs(y - current_y) > 1.5:
                    if buf:
                        lines.append(" ".join(buf))
                    buf = [word]
                    current_y = y
                else:
                    buf.append(word)
            if buf:
                lines.append(" ".join(buf))
            return "\n".join(lines)
        except Exception:
            return ""

    def _extract_text_ocr(page) -> str:
        if not HAS_OCR:
            return ""
        try:
            pix = page.get_pixmap(dpi=200, alpha=False)
            img = Image.frombytes("RGB", (pix.width, pix.height), pix.samples)
            txt = pytesseract.image_to_string(img) or ""
```

```
            return txt
        except Exception:
            return ""

    def _looks_scanned_or_empty(text: str, page) -> bool:
        # Heuristic: very few alphabetic characters OR page contains large images
        alpha = sum(ch.isalpha() for ch in text)
        if alpha < 50:
            return True
        try:
            return len(page.get_images(full=True)) > 0 and len(text) < 200
        except Exception:
            return False

    # ------------------------------------------------------------------------
    # 1) extract_pdf_text — robust multi-mode extraction + cleaning + tagging
    # ------------------------------------------------------------------------
    def extract_pdf_text(pdf_path: Path) -> Tuple[str, List[Dict]]:
        """
        Extract text per page using PyMuPDF, clean it, tag sections.
        Returns:
          full_clean_text (str),
          pages: [
            {"page": int, "text": str, "section": str, "doc_name": str},
            ...
          ]
        """
        doc = fitz.open(pdf_path)
        pages: List[Dict] = []
        try:
            for pno in range(len(doc)):
                page = doc[pno]

                # 1) try multiple extraction modes in order
                raw = _extract_text_text_mode(page)
                if not raw.strip():
                    raw = _extract_text_blocks_mode(page)
                if not raw.strip():
```

```
                raw = _extract_text_words_mode(page)

            # 2) OCR fallback only if needed
            if _looks_scanned_or_empty(raw, page):
                ocr_txt = _extract_text_ocr(page)
                if ocr_txt.strip():
                    raw = ocr_txt

            # 3) clean & normalize lines
            clines = clean_lines(raw.splitlines())
            text = _norm_unicode("\n".join(clines))

            # 4) keep only non-empty
            if text:
                section = detect_section(text)
                pages.append({
                    "page": int(pno + 1),
                    "text": text,
                    "section": section,
                    "doc_name": pdf_path.stem
                })
    finally:
        doc.close()

    # 5) ensure sorted by page
    pages.sort(key=lambda d: d["page"])

    # 6) join full clean text
    full = "\n\n".join(p["text"] for p in pages)
    return full, pages

# ------------------------------------------------------------
# 2) write_clean_text — validation + sorting + atomic save
# ------------------------------------------------------------
def write_clean_text(doc_name: str, full_text: str, pages: List[Any]):
    """
    Save cleaned full text and per-page JSON.
    Accepts pages as either:
```

```
  - List[Tuple[int, str]]  -> (page_no, page_text)
  - List[Dict]             -> {"page":..., "text":..., ...}
"""
CLEAN_DIR.mkdir(parents=True, exist_ok=True)

# normalize full text (light)
full_text = _norm_unicode(full_text)
tmp_full = CLEAN_DIR / f".{doc_name}.txt.tmp"
(CLEAN_DIR / f"{doc_name}.txt").write_text(full_text, encoding="utf-8")

# normalize pages to a list of dicts
per_page_norm: List[Dict[str, Any]] = []
if pages and isinstance(pages[0], dict):
    for d in pages:
        # coerce types; keep extra metadata if any
        rec = {
            "page": int(d.get("page", 0)),
            "text": str(d.get("text", "")),
            **{k: v for k, v in d.items() if k not in {"page", "text"}}
        }
        per_page_norm.append(rec)
else:
    for (p, t) in pages:
        per_page_norm.append({"page": int(p), "text": str(t)})

# sort & dedupe by page (keep first occurrence)
per_page_norm.sort(key=lambda r: r.get("page", 0))
seen = set()
unique_pages = []
for r in per_page_norm:
    pg = r.get("page", 0)
    if pg in seen:
        continue
    seen.add(pg)
    # light normalize text field too
    r["text"] = _norm_unicode(r.get("text", ""))
    unique_pages.append(r)
```

```
    # atomic write
    tmp_json = CLEAN_DIR / f".{doc_name}.pages.json.tmp"
    final_json = CLEAN_DIR / f"{doc_name}.pages.json"
    tmp_json.write_text(json.dumps(unique_pages, ensure_ascii=False, indent=2), encoding="utf-8")
    tmp_json.replace(final_json)
```

```
def detect_section(text: str) -> str:
    for section, pattern in SECTION_PATTERNS.items():
        if re.search(pattern, text, flags=re.I):
            return section
    return "other"

SECTION_PATTERNS = {
    # Balance sheet synonyms
    "balance_sheet": r"\b("
        r"balance\s+sheet"
        r"|statement\s+of\s+financial\s+position"
        r"|statement\s+of\s+assets\s+and\s+liabilities"
        r")\b",

    # Income statement / P&L synonyms
    "income_statement": r"\b("
        r"income\s+statement"
        r"|profit\s+and\s+loss"
        r"|p&l"
        r"|statement\s+of\s+(operations|earnings|comprehensive\s+income)"
        r")\b",

    # Cash flow synonyms
    "cash_flow": r"\b("
        r"cash\s+flow(\s+statement)?"
        r"|statement\s+of\s+cash\s+flows"
        r"|funds\s+flow\s+statement"
        r"|sources\s+and\s+uses\s+of\s+cash"
        r")\b",
```

```
        # Management discussion & analysis
        "mdna": r"\b("
            r"management\s+discussion\s+and\s+analysis"
            r"|md&a"
            r"|operating\s+and\s+financial\s+review"
            r"|management’s?\s+report"
            r"|business\s+overview"
            r")\b",

        # Notes to financial statements
        "notes": r"\b("
            r"notes?\s+(to\s+(the\s+)?)?(financial\s+statements|accounts)"
            r"|explanatory\s+notes"
            r"|footnotes"
            r")\b",

        # Corporate governance / directors
        "governance": r"\b("
            r"corporate\s+governance"
            r"|board\s+report"
            r"|directors?'?\s+report"
            r")\b",

        # Auditor’s reports
        "audit_report": r"\b("
            r"auditors?'?\s+report"
            r"|independent\s+auditors?'?\s+report"
            r")\b",

        # Risk and ESG
        "risk": r"\b(risk\s+management|principal\s+risks?|risk\s+factors)\b",
        "esg": r"\b(environmental\s*,?\s*social\s*(and|&)\s*governance|sustainability\s+report)\b",

        # Shareholder information
        "shareholder_info": r"\b("
            r"shareholder\s+information"
            r"|investor\s+information"
            r"|share\s+capital"
```

```
        r")\b"
}


def rough_section_indices(text: str) -> Dict[str, List[int]]:
    idx = {}
    low = text.lower()
    for name, pat in SECTION_PATTERNS.items():
        idx[name] = [m.start() for m in re.finditer(pat, low)]
    return idx

def segment_sections(text: str) -> Dict[str, str]:
    indices = rough_section_indices(text)
    all_starts = []
    for sec, starts in indices.items():
        for s in starts:
            all_starts.append((s, sec))
    if not all_starts:
        return {"full_report": text}

    all_starts.sort(key=lambda x: x[0])
    result = {}
    for i, (start, sec) in enumerate(all_starts):
        end = all_starts[i+1][0] if i+1 < len(all_starts) else len(text)
        if sec not in result:  # keep first occurrence only
            result[sec] = text[start:end].strip()
    if not result:
        result["full_report"] = text
    return result
```

Chunking (100-word & 400-word windows with overlap)

```
import re, os, json, bisect
from typing import List, Dict, Tuple, Any
```

```
# ---------- helpers ----------
_SPACE_RE = re.compile(r"\s+")
def _collapse_ws(s: str) -> str:
    return _SPACE_RE.sub(" ", s).strip()

def _is_sentence_end_token(tok: str) -> bool:
    # crude but effective: token ends with ., !, or ?
    return tok.endswith(".") or tok.endswith("!") or tok.endswith("?")

# ---------- chunking ----------
def chunk_words(words, chunk_size, overlap=20):
    """
    Safer windowing:
      - clamps invalid params
      - favors sentence boundary within a small tail window
      - avoids tiny trailing fragments
    Output unchanged: yields (i, j, " ".join(words[i:j]))
    """
    if chunk_size <= 0:
        return
    if overlap < 0:
        overlap = 0
    if overlap >= chunk_size:
        # keep at least one-token advance to avoid infinite loop
        overlap = max(0, chunk_size - 1)

    n = len(words)
    if n == 0:
        return

    i = 0
    # search window to find a sentence end near the chunk tail
    tail_seek = max(5, min(20, chunk_size // 5))

    while i < n:
        j = min(i + chunk_size, n)

        # try to end on a sentence boundary if possible within [j - tail_seek, j]
```

```
            if j - i > 8:  # only if chunk has a bit of content
                start_seek = max(i + 1, j - tail_seek)
                best = None
                for k in range(j - 1, start_seek - 1, -1):
                    if _is_sentence_end_token(words[k]):
                        best = k + 1  # include the sentence-end token
                        break
                if best and best > i + 5:  # avoid making too-small chunks
                    j = best

            # ensure non-empty slice
            if j <= i:
                j = min(i + 1, n)

            text = _collapse_ws(" ".join(words[i:j]))
            if text:
                yield (i, j, text)

            if j == n:
                break

            # stride with overlap
            i = max(j - overlap, i + 1)

            # avoid producing a very tiny trailing fragment (< ~10% of chunk_size)
            if n - i < max(5, chunk_size // 10):
                # extend previous window to include the tail (skip emitting tiny)
                break

    def make_chunks(doc_name, full_text, pages, sizes=(100, 400)):
        """
        Build overlapping chunks of given sizes from pages.
        Supports both:
          - pages = [(page_no, text), ...]
          - pages = [{"page":..., "text":...}, ...]
        Output shape unchanged.
        """
        chunks_by_size = {s: [] for s in sizes}
```

```
# Normalize pages into (page_no:int, text:str)
norm_pages: List[Tuple[int, str]] = []
for item in pages:
    if isinstance(item, dict):
        p = int(item.get("page", 0) or 0)
        t = str(item.get("text", "") or "")
    else:
        p, t = item
        p = int(p); t = str(t)
    if p > 0 and t:
        norm_pages.append((p, t))
# sort by page just in case
norm_pages.sort(key=lambda x: x[0])

# Precompute cumulative word spans per page: [ (page, start_idx_in_full, end_idx_in_full) ]
page_word_spans = []
cum = 0
page_starts = []  # for bisect
for p, t in norm_pages:
    wc = len(t.split())
    start, end = cum, cum + wc
    page_word_spans.append((p, start, end))
    page_starts.append(start)
    cum = end

words = full_text.split()
total_words = len(words)

for size in sizes:
    # stride ~= half window (classic overlap), but chunk_words will clamp nicely
    stride = max(1, size // 2)

    for (i, j, text) in chunk_words(words, chunk_size=size, overlap=size - stride):
        # map [i, j) to covered pages using binary search on starts
        if page_word_spans:
            # find first page whose end > i
            # we'll expand linearly outward due to typically small overlaps
```

```
                    # but use bisect to anchor near the right region
                    idx = bisect.bisect_right(page_starts, i) - 1
                    if idx < 0:
                        idx = 0
                    covered = []
                    # collect forward while page start < j
                    k = idx
                    while k < len(page_word_spans):
                        pg, s, e = page_word_spans[k]
                        if s >= j:
                            break
                        if e > i and s < j:
                            covered.append(pg)
                        k += 1
                    # dedupe & preserve order
                    seen = set()
                    covered_pages = []
                    for pg in covered:
                        if pg not in seen:
                            covered_pages.append(pg)
                            seen.add(pg)
                else:
                    covered_pages = []

                chunks_by_size[size].append({
                    "doc_name": doc_name,
                    "chunk_size": size,
                    "chunk_id": f"{doc_name}_{size}_{i//stride}",
                    "text": text,
                    "pages_approx": covered_pages,
                })

        return chunks_by_size

    # ---------- saving ----------
    def save_chunks(all_chunks: Dict[int, list]):
        """
        Writes:
```

```
  - PROC_DIR/chunks_<size>.jsonl  (streamed)
  - PROC_DIR/chunks_<size>.json   (pretty, full list)
Output unchanged; now uses atomic replace.
"""
PROC_DIR.mkdir(parents=True, exist_ok=True)
for sz, chs in all_chunks.items():
    # JSONL (stream)
    jsonl_path = PROC_DIR / f"chunks_{sz}.jsonl"
    tmp_jsonl  = PROC_DIR / f".chunks_{sz}.jsonl.tmp"
    with open(tmp_jsonl, "w", encoding="utf-8") as f:
        for c in chs:
            # ensure minimal normalization
            if "text" in c:
                c["text"] = _collapse_ws(c["text"])
            f.write(json.dumps(c, ensure_ascii=False) + "\n")
    os.replace(tmp_jsonl, jsonl_path)

    # JSON (pretty)
    pretty_path = PROC_DIR / f"chunks_{sz}.json"
    tmp_pretty  = PROC_DIR / f".chunks_{sz}.json.tmp"
    with open(tmp_pretty, "w", encoding="utf-8") as f:
        json.dump(chs, f, ensure_ascii=False, indent=2)
    os.replace(tmp_pretty, pretty_path)
```

Run on All Raw PDFs

```
import json, time, gc
from typing import Dict, List

# Toggle to force reprocessing even if clean artifacts already exist
FORCE_REBUILD = False

pdfs = sorted(RAW_DIR.glob("*.pdf"))
assert len(pdfs) > 0, "No PDFs found in data/raw/. Please add your annual reports."
```

```
all_chunks_100, all_chunks_400 = [], []
doc_stats: List[Dict] = []

def _already_processed(doc_name: str) -> bool:
    txt_ok   = (CLEAN_DIR / f"{doc_name}.txt").exists()
    pages_ok = (CLEAN_DIR / f"{doc_name}.pages.json").exists()
    secs_ok  = (CLEAN_DIR / f"{doc_name}.sections.json").exists()
    return txt_ok and pages_ok and secs_ok

for idx, pdf in enumerate(pdfs, 1):
    t0 = time.time()
    doc_name = pdf.stem
    print(f"[{idx}/{len(pdfs)}] Processing {pdf.name} ...")

    try:
        # Skip if artifacts exist and not forcing rebuild
        if not FORCE_REBUILD and _already_processed(doc_name):
            # Load for stats if needed
            full_text = (CLEAN_DIR / f"{doc_name}.txt").read_text(encoding="utf-8", errors="ignore")
            pages = json.loads((CLEAN_DIR / f"{doc_name}.pages.json").read_text(encoding="utf-8", errors="ignore"))
            # Build chunks anyway (ensures combined chunk files reflect current params)
            chunks_by_size = make_chunks(doc_name, full_text, pages, sizes=(100, 400))
        else:
            # Extract & clean
            full_text, pages = extract_pdf_text(pdf)

            # Persist cleaned text + per-page JSON
            write_clean_text(doc_name, full_text, pages)

            # Rough section splits (narrative-level segmentation)
            sections = segment_sections(full_text)
            (CLEAN_DIR / f"{doc_name}.sections.json").write_text(
                json.dumps(sections, ensure_ascii=False, indent=2), encoding="utf-8"
            )

            # Build chunks (100 & 400)
            chunks_by_size = make_chunks(doc_name, full_text, pages, sizes=(100, 400))
```

```
            # Append to global pools
            all_chunks_100.extend(chunks_by_size[100])
            all_chunks_400.extend(chunks_by_size[400])

            # Per-doc stats
            n_pages = len(pages) if isinstance(pages, list) else 0
            n_chars = len(full_text)
            doc_stats.append({
                "doc": pdf.name,
                "pages": n_pages,
                "chars": n_chars,
                "chunks_100": len(chunks_by_size[100]),
                "chunks_400": len(chunks_by_size[400]),
                "elapsed_s": round(time.time() - t0, 2)
            })

            # Light sanity logs
            print(f"  -> pages={n_pages}, chars={n_chars:,}, "
                  f"100w={len(chunks_by_size[100])}, 400w={len(chunks_by_size[400])}, "
                  f"time={doc_stats[-1]['elapsed_s']}s")

        except Exception as e:
            # Don't abort entire batch on a single bad PDF
            print(f"[WARN] Failed on {pdf.name}: {e}")
        finally:
            # Help free memory between documents
            gc.collect()

    # Optional: de-duplicate chunk_ids defensively (shouldn't be necessary, but safe)
    def _dedupe_by_chunk_id(chs: List[Dict]) -> List[Dict]:
        seen = set()
        out = []
        for c in chs:
            cid = c.get("chunk_id")
            if cid in seen:
                continue
            seen.add(cid)
            out.append(c)
```

```
    return out

all_chunks_100 = _dedupe_by_chunk_id(all_chunks_100)
all_chunks_400 = _dedupe_by_chunk_id(all_chunks_400)

# Save combined chunk files (unchanged API)
save_chunks({100: all_chunks_100, 400: all_chunks_400})

# Stats & summary
print("\n[STATS] Documents processed:", [p.name for p in pdfs])
print("[STATS] 100-word chunks:", len(all_chunks_100))
print("[STATS] 400-word chunks:", len(all_chunks_400))
if doc_stats:
    # pretty per-doc summary
    print("\n[PER-DOC]")
    for s in doc_stats:
        print(f" - {s['doc']}: pages={s['pages']}, chars={s['chars']:,}, "
              f"100w={s['chunks_100']}, 400w={s['chunks_400']}, time={s['elapsed_s']}s")

print("\n[OK] Outputs saved to:", PROC_DIR.resolve())
```

```
[1/2] Processing annual-report-2024.pdf ...
  -> pages=352, chars=1,036,537, 100w=3414, 400w=950, time=0.76s
[2/2] Processing infosys-ar-25.pdf ...
  -> pages=368, chars=1,094,340, 100w=3558, 400w=825, time=0.68s

[STATS] Documents processed: ['annual-report-2024.pdf', 'infosys-ar-25.pdf']
[STATS] 100-word chunks: 6304
[STATS] 400-word chunks: 1576

[PER-DOC]
 - annual-report-2024.pdf: pages=352, chars=1,036,537, 100w=3414, 400w=950, time=0.76s
 - infosys-ar-25.pdf: pages=368, chars=1,094,340, 100w=3558, 400w=825, time=0.68s

[OK] Outputs saved to: /content/drive/MyDrive/RAG-FT-DATA/processed
```

Quick Peek / Sanity Check

```
# Show a few cleaned lines and a couple of chunks for verification
txt_files = sorted(CLEAN_DIR.glob("*.txt"))
print("Cleaned text files:", [p.name for p in txt_files][:5])

sample_txt = txt_files[0].read_text(encoding="utf-8").splitlines()[:15]
# sample_txt = txt_files[0].read_text(encoding="utf-8").splitlines()
print("\nSample cleaned lines:\n", "\n".join(sample_txt))

import json
sample_100 = json.loads((PROC_DIR / "chunks_100.json").read_text(encoding="utf-8"))[:10]
pd.DataFrame(sample_100)[["chunk_id","pages_approx","text"]].head(20)
```

```
Cleaned text files: ['annual-report-2024.txt', 'infosys-ar-25.txt']

Sample cleaned lines:
 Generative AI and You

We barely saw it happen. AI walking into our lives. Through
the ads that follow us on social media. The personalized
pick of movies and shows. Our cars. The maps helping us
navigate. Right there in our hands — our super-powerful
phones. And now, it’s happening again. This time with
generative AI. In the form of handy tools — like ChatGPT,
MetaAI and Stable Diffusion — that pique our imagination,
and stoke our curiosity.
Generative AI technology’s path into enterprises too has
been just as accelerated and enthusiastic, supported by
an exponential increase in investments. While almost
every enterprise has started working with generative
AI, their struggles with data readiness and concerns
```

	chunk_id	pages_approx	text
0	annual-report-2024_100_0	[1, 2]	Generative AI and You We barely saw it happen....
1	annual-report-2024_100_1	[2]	exponential increase in investments. While alm...
2	annual-report-2024_100_2	[2]	In the months to follow, we believe, some ongo...
3	annual-report-2024_100_3	[2]	them. For example, supporting and personalizin...
4	annual-report-2024_100_4	[2]	for more pervasive automation, will accelerate...
5	annual-report-2024_100_5	[2]	Embracing AI responsibly also means keeping tr...
6	annual-report-2024_100_6	[2]	and sustainability impact too. We are factorin...
7	annual-report-2024_100_7	[2, 3]	next as generative AI paves the path partner o...
8	annual-report-2024_100_8	[2, 3]	tools. Microsoft Corporation is a technology c...
9	annual-report-2024_100_9	[3]	especially generative AI, to heighten their op...

```
OUT_QA_JSONL = ROOT / "qa_pairs.jsonl"
OUT_QA_CSV   = ROOT / "qa_pairs.csv"
```

```
import json, re, unicodedata
from pathlib import Path
from typing import Dict

# ---------- precompiled regexes ----------
_SPACE_RE          = re.compile(r"\s+")
_YEAR_RE           = re.compile(r"\b(20\d{2})\b")
_FY_RE             = re.compile(r"\bfy\s*'?(\d{2,4})\b", re.I)  # FY24 / FY 2024
# Proper-noun span (1–4 tokens), allows & and dots within tokens (e.g., "Procter & Gamble", "L&T", "T. Rowe")
_COMPANY_SPAN_RE   = re.compile(r"\b([A-Z][A-Za-z.&\-']+(?:\s+[A-Z][A-Za-z.&\-']+){0,3})\b")
# Common corporate suffixes to prefer if present
_COMPANY_SUFFIX_RE = re.compile(r"\b(limited|ltd\.?|plc|inc\.?|corp\.?|corporation|company|co\.?)\b", re.I)

def _norm_unicode(s: str) -> str:
    # Normalize and strip BOM/NBSP and excessive spaces
    s = unicodedata.normalize("NFKC", s).replace("\uFEFF", "").replace("\u00A0", " ")
    return s

# ---------- 1) load_clean_texts ----------
def load_clean_texts(clean_dir: Path) -> Dict[str, str]:
    """
    Load *.txt from clean_dir into {doc_stem: text}, with unicode normalization.
    Skips unreadable files rather than raising.
    """
    texts: Dict[str, str] = {}
    # deterministic order
    for p in sorted(clean_dir.glob("*.txt")):
        try:
            raw = p.read_text(encoding="utf-8", errors="ignore")
            raw = _norm_unicode(raw)
            texts[p.stem] = raw
        except Exception:
            # skip bad file but continue
            continue
```

```
    return texts

# ---------- 2) load_sections ----------
def load_sections(clean_dir: Path) -> Dict[str, Dict[str, str]]:
    """
    Load *.sections.json into {doc_stem: <json_object>}.
    Accepts any JSON type inside (list/dict), but still returns a dict mapping file stems.
    Invalid JSON files are skipped silently.
    """
    sections: Dict[str, Dict[str, str]] = {}
    for p in sorted(clean_dir.glob("*.sections.json")):
        try:
            data = json.loads(p.read_text(encoding="utf-8", errors="ignore"))
            # keep exactly the same outward type: a dict mapping stem->data (whatever data is)
            sections[p.stem.replace(".sections", "")] = data
        except Exception:
            continue
    return sections

# ---------- 3) normalize_spaces ----------
def normalize_spaces(s: str) -> str:
    """
    Collapse whitespace to single spaces and strip.
    """
    return _SPACE_RE.sub(" ", s).strip()

# ---------- 4) guess_company_from_text ----------
def guess_company_from_text(doc_name: str, text: str) -> str:
    """
    Heuristic guess using top-of-document lines.
    Prefers spans that include a company suffix (Ltd, PLC, Inc, …).
    Falls back to longest proper-noun span; finally to a neat doc_name.
    """
    # consider first ~40 lines for titles/headers
    head = "\n".join(text.splitlines()[:40])
    head = _norm_unicode(head)

    # collect candidates (dedup preserve order)
```

```
        seen = set()
        cands = []
        for m in _COMPANY_SPAN_RE.finditer(head):
            span = m.group(1).strip()
            if span not in seen:
                seen.add(span)
                cands.append(span)

        if not cands:
            # tidy fallback from filename: underscores/dashes -> spaces, title-cased
            fallback = re.sub(r"[_\-]+", " ", doc_name).strip()
            return fallback.title() if fallback else doc_name

        # 1) prefer candidate that contains a corporate suffix
        for c in cands:
            if _COMPANY_SUFFIX_RE.search(c):
                return c

        # 2) prefer the longest candidate (more tokens/chars) near "Annual Report" wording
        annual_hit = re.search(r"(annual|integrated).{0,15}report", head, flags=re.I)
        if annual_hit:
            # pick candidate closest (by index distance) to the annual report phrase
            pos = annual_hit.start()
            nearest = min(cands, key=lambda s: abs(head.find(s) - pos) if head.find(s) >= 0 else 1_000_000)
            return nearest

        # 3) otherwise, longest by length
        return max(cands, key=len)

    # ---------- 5) find_years ----------
    def find_years(text: str):
        """
        Return sorted unique years found in text, augmenting with FY patterns.
        Keeps the original return type (List[int]).
        """
        text_norm = _norm_unicode(text).lower()

        # direct years like 2023, 2024
```

```
    years = {int(y) for y in _YEAR_RE.findall(text_norm) if 2000 <= int(y) <= 2100}

    # FY abbreviations like FY24 → 2024, FY'23 → 2023
    for fy in _FY_RE.findall(text_norm):
        try:
            n = int(fy)
            if n < 100:      # FY24 -> 2024; assume 2000s
                n = 2000 + n
            if 2000 <= n <= 2100:
                years.add(n)
        except Exception:
            pass

    return sorted(years)
```

Patterns & Extraction

```
import re
from typing import List, Dict, Any

# --- Amount/units patterns (improved but same variable names) ---
CURRENCY = r"(?:₹|Rs\.?|INR|USD|EUR|\$|€|£)"
NUM_CORE = r"(?:\d{1,3}(?:[,\s]\d{3})*(?:\.\d+)?|\d+(?:\.\d+)?)"
# allow accounting negatives: (1,234) or -1,234 or −1,234
NUM_SIGNED = rf"(?:\(?-?\s*{NUM_CORE}\)?|−\s*{NUM_CORE})"
UNITS = r"(?:crore|cr\.?|cr|lakh|million|mn|billion|bn|m|bn)"
# amount forms: [currency] number [unit]?  OR  currency unit number
AMT_A = rf"(?:{CURRENCY}\s*)?{NUM_SIGNED}(?:\s*{UNITS})?"
AMT_B = rf"(?:{CURRENCY}\s*(?:{UNITS})\s*{NUM_SIGNED})"
AMOUNT = rf"(?:{AMT_A}|{AMT_B})"

# percentages (to exclude where irrelevant)
PCT = r"(?:\d+(?:\.\d+)?\s*%)"

# More generous metric aliases; keep keys unchanged
```

```
METRIC_PATTERNS = {
    "revenue": rf"\b("
               r"revenue"
               r"|total\s+revenue"
               r"|net\s+sales"
               r"|sales"
               r"|income\s+from\s+operations"
               r")\b[\s:\-–]*"
               rf"(?!{PCT})"  # avoid % values like “revenue growth 12%”
               rf"{AMOUNT}",
    "net_income": rf"\b("
                  r"net\s+(?:income|profit)"
                  r"|profit\s+after\s+tax"
                  r"|profit\s+for\s+the\s+year"
                  r"|PAT\b"
                  r")\b[\s:\-–]*"
                  rf"(?!{PCT})"
                  rf"{AMOUNT}",
    "ebitda": rf"\b("
              r"EBITDA"
              r"|earnings\s+before\s+interest,\s*tax,\s*depreciation\s+and\s+amortization"
              r")\b[\s:\-–]*"
              rf"(?!{PCT})"
              rf"{AMOUNT}",
    "eps": rf"\b("
           r"EPS\b"
           r"|earnings\s+per\s+share"
           r")\b[\s:\-–]*"
           rf"(?:{AMOUNT}|{NUM_SIGNED})",   # EPS often has no currency/unit
    "cash_flow": rf"\b("
                 r"net\s+cash\s+from\s+operating\s+activities"
                 r"|cash\s+generated\s+from\s+operations"
                 r"|operating\s+cash\s+flow"
                 r")\b[\s:\-–]*"
                 rf"(?!{PCT})"
                 rf"{AMOUNT}",
    "assets": rf"\b("
              r"total\s+assets"
```

```
                r"|sum\s+total\s+assets"
                r"|aggregate\s+assets"
                r")\b[\s:\—]*"
                rf"(?!{PCT})"
                rf"{AMOUNT}",
    "liabilities": rf"\b("
                   r"total\s+liabilities"
                   r"|aggregate\s+liabilities"
                   r")\b[\s:\—]*"
                   rf"(?!{PCT})"
                   rf"{AMOUNT}",
}

# --- Helpers (same function names/returns preserved) ---

_INSIDE_NUM_SPACE = re.compile(r"(?<=\d)\s+(?=\d)")  # join "4 189" -> "4189"
_MULTI_WS = re.compile(r"\s+")
_YEAR = re.compile(r"\b(20\d{2})\b")
_FY = re.compile(r"\bfy\s*'?\s*(\d{2,4})\b", re.I)

def normalize_spaces(s: str) -> str:
    return _MULTI_WS.sub(" ", s).strip()

def _clean_number_artifacts(s: str) -> str:
    # repair split thousands and normalize spaces around signs/parentheses
    s = _INSIDE_NUM_SPACE.sub("", s)
    s = s.replace(" )", ")").replace("( ", "(").replace("- ", "-").replace("- ", "-")
    return normalize_spaces(s)

def clean_amount(val: str) -> str:
    v = normalize_spaces(val)
    # normalize common currency tokens spacing
    v = v.replace("USD", "USD ").replace("INR", "INR ").replace("Rs.", "Rs ").replace("Rs", "Rs ")
    v = v.replace("₹", "₹ ")
    # normalize unit shorthands
    v = re.sub(r"\bmn\b", "million", v, flags=re.I)
    v = re.sub(r"\bbn\b", "billion", v, flags=re.I)
    v = re.sub(r"\bcr\.?\b", "crore", v, flags=re.I)
```

```
        v = re.sub(r"\bm\b(?![a-z])", "million", v, flags=re.I)  # plain 'm' as million (common in ARs)
        # fix split digits like "4 189"
        v = _clean_number_artifacts(v)
        # remove stray space after currency if double-spaced
        v = re.sub(r"(₹|Rs|USD|INR)\s{2,}", r"\1 ", v, flags=re.I)
        return v.strip()

    def _year_from_near(text: str) -> int:
        # prefer explicit year; fallback to FY mapping (FY24 -> 2024)
        y = _YEAR.search(text)
        if y:
            return int(y.group(1))
        fy = _FY.search(text)
        if fy:
            n = int(fy.group(1))
            if n < 100:  # FY24 -> 2024
                n += 2000
            return n if 2000 <= n <= 2100 else None
        return None

    def extract_metric_sentences(text: str, metric: str, pattern: str) -> List[Dict[str, Any]]:
        out = []
        # Wider context windows for reliability
        LEFT_CTX, RIGHT_CTX = 200, 200
        for m in re.finditer(pattern, text, flags=re.IGNORECASE):
            span = m.span()
            start = max(0, span[0] - LEFT_CTX)
            end   = min(len(text), span[1] + RIGHT_CTX)
            context = normalize_spaces(text[start:end])
            raw = normalize_spaces(m.group(0))

            # find the *first* amount in raw; if not found, try small extension rightwards
            amt_m = re.search(AMOUNT, raw, flags=re.IGNORECASE)
            if not amt_m:
                ext_end = min(len(text), span[1] + 40)
                tail = normalize_spaces(text[span[0]:ext_end])
                amt_m = re.search(AMOUNT, tail, flags=re.IGNORECASE)
```

```
            amount = clean_amount(amt_m.group(0)) if amt_m else raw

            # pick nearby year (including FYxx)
            near = text[max(0, span[0]-100): min(len(text), span[1]+100)]
            year = _year_from_near(near)

            # exclude pure percentages that slipped through (belt-and-braces)
            if re.fullmatch(r"\s*\d+(?:\.\d+)?\s*%\s*", amount):
                continue

            out.append({
                "metric": metric,
                "amount": amount,
                "year": year,
                "raw": raw,
                "context": context
            })
        return out

    def extract_all_metrics(text: str) -> List[Dict[str, Any]]:
        results = []
        for metric, pat in METRIC_PATTERNS.items():
            results.extend(extract_metric_sentences(text, metric, pat))
        return results
```

Broaden metric patterns (percentages, headcount, dividends, etc.)

```
import re

# --- Base number/currency pieces (keep original names, just stronger) ---
CURRENCY = r"(?:₹|Rs\.?|INR|USD|EUR|\$|€|£)"
NUM_CORE = r"(?:\d{1,3}(?:[,\s]\d{3})*(?:\.\d+)?|\d+(?:\.\d+)?)"
# allow negatives and accounting parentheses: (1,234), -1,234, −1,234
NUM = rf"(?:\(?-?\s*{NUM_CORE}\)?|−\s*{NUM_CORE})"
```

```
# --- Indian & global units (your name retained) ---
INDIAN_UNITS = r"(?:crore|cr\.?|cr|lakh|lakhs|million|mn|billion|bn|thousand|k|m)"

# --- Amount pattern (currency optional, units optional). Keep name AMOUNT. ---
AMOUNT = rf"(?:{CURRENCY}\s*)?{NUM}(?:\s*{INDIAN_UNITS})?"

# percentages (for margin-type metrics)
PCT = r"(?:\d+(?:\.\d+)?\s*%)"

# --- Extra patterns: keep same dict name/keys; broaden aliases & robustness ---
# Note: some items (headcount, EPS/dividend) can be plain numbers without currency.
EXTRA_PATTERNS = {
    "operating_margin":  rf"\b(operating\s+margin|EBIT\s*margin)\b(?:\s+(?:was|at|of|stood\s+at))?[\s:\—]*{PCT}",
    "net_margin":        rf"\b(net\s+margin)\b(?:\s+(?:was|at|of|stood\s+at))?[\s:\—]*{PCT}",

    # headcount usually not currency; allow large ints with commas and optional 'employees'
    "headcount":         rf"\b(headcount|number\s+of\s+employees|employees)\b[\s:\—]*(?:approximately\s+|around\s-
                         rf"(?:\d{{1,3}}(?:,\d{{3}})+|\d+)\b(?:\s+employees)?",

    # dividend per share can be currency or plain number; include 'final/interim/total dividend'
    "dividend":          rf"\b((?:final|interim|total)\s+dividend|dividend(?:\s+per\s+share)?|dps)\b[\s:\—]*"
                         rf"(?:{AMOUNT}|{NUM})",

    # equity synonyms
    "equity":            rf"\b(total\s+equity|shareholders?'?\s*funds|net\s+worth|shareholders?'?\s*equity)\b[\s:\-

    # cash & cash equivalents synonyms (ampersand or 'and')
    "cash_and_cash_eq":  rf"\b(cash\s+(?:&|and)\s+cash\s+equivalents)\b[\s:\—]*{AMOUNT}",

    # R&D expenses variants
    "rd_expense":        rf"\b(research\s+and\s+development\s+expenses|R&D\s+expenses?)\b[\s:\—]*{AMOUNT}",

    # capex/opex broader aliases
    "capex":             rf"\b(capex|capital\s+expenditure[s]?)\b[\s:\—]*{AMOUNT}",
    "opex":              rf"\b(operating\s+expenses|opex|selling,\s+general\s+and\s+administrative\s+expenses|SG&A
}

# --- Finance metrics: keep dict name & keys, broaden value capturing w/ units ---
```

```
FINANCE_METRICS = {
    "total assets": rf"\btotal\s+assets\b[\s:\—]*{AMOUNT}",
    "net profit":   rf"\b(net\s+profit|profit\s+after\s+tax|profit\s+for\s+the\s+year|PAT)\b[\s:\—]*{AMOUNT}",
    "revenue":      rf"\b((?:total\s+)?revenue|net\s+sales|sales|income\s+from\s+operations)\b[\s:\—]*{AMOUNT}",
}

# --- Merge into your existing METRIC_PATTERNS (order fixed so AMOUNT is already defined) ---
METRIC_PATTERNS.update(EXTRA_PATTERNS)
METRIC_PATTERNS.update(FINANCE_METRICS)
```

Add textual extractors (CEO, CFO, Auditor, HQ, segments)

```
# honorifics to strip from captured names
_HONORIFICS_RE = re.compile(r"^(mr\.?|mrs\.?|ms\.?|dr\.?|shri|smt\.?)\s+", re.I)
_MULTI_WS_RE   = re.compile(r"\s+")
_END_PUNC_RE   = re.compile(r"[,\.;:]+$")

def _name_clean(s: str) -> str:
    s = s.strip()
    s = _HONORIFICS_RE.sub("", s)
    s = _MULTI_WS_RE.sub(" ", s)
    s = _END_PUNC_RE.sub("", s)
    # If name looks like ALL CAPS, title-case it
    letters = [ch for ch in s if ch.isalpha()]
    if letters and sum(ch.isupper() for ch in letters) / len(letters) > 0.8:
        s = s.title()
    return s

def extract_textual_facts(text: str) -> List[Dict]:
    facts = []
    seen = set()  # (type, value) de-dupe
    t = text  # alias

    # --- CEO / CFO (broader phrasing) ---
    # Captures names like "A. B. Lastname", allows 1–4 tokens
    PERSON = r"([A-Z][A-Za-z\.\-']+(?:\s+[A-Z][A-Za-z\.\-']+){0,3})"
```

```
        ceo_patterns = [
            rf"\b(Chief\s+Executive\s+Officer|CEO)\b[:\—–]?\s*{PERSON}",
            rf"\b(MD\s*&\s*CEO|CEO\s*&\s*MD|Managing\s+Director\s*&\s*CEO|Chief\s+Executive\s+and\s+Managing\s+Directo
        ]
        cfo_patterns = [
            rf"\b(Chief\s+Financial\s+Officer|CFO)\b[:\—–]?\s*{PERSON}",
        ]

        for pat in ceo_patterns:
            for m in re.finditer(pat, t, flags=re.I):
                person = _name_clean(m.group(m.lastindex))
                ctx = normalize_spaces(t[max(0, m.start()-160): m.end()+160])
                if person and ("ceo", person) not in seen:
                    facts.append({"type": "ceo", "value": person, "context": ctx})
                    seen.add(("ceo", person))

        for pat in cfo_patterns:
            for m in re.finditer(pat, t, flags=re.I):
                person = _name_clean(m.group(m.lastindex))
                ctx = normalize_spaces(t[max(0, m.start()-160): m.end()+160])
                if person and ("cfo", person) not in seen:
                    facts.append({"type": "cfo", "value": person, "context": ctx})
                    seen.add(("cfo", person))

        # --- Auditor (firm names; stop at newline; allow LLP/LLP., & Co., etc.) ---
        # Examples: "XYZ & Co. LLP, Chartered Accountants"
        auditor_pat = (
            r"\b("
            r"Statutory\s+Auditors?|Independent\s+Auditors?|Auditor"
            r")\b[:\—–]?\s*"
            r"([A-Z0-9&][A-Za-z0-9&\s\.,'/-]{3,120}?)"
            r"(?=$|\n)"  # stop at end-of-line
        )
        for m in re.finditer(auditor_pat, t, flags=re.I):
            val = normalize_spaces(m.group(2))
            # trim trailing qualifiers if they repeat heavily after comma
            val = re.sub(r"\s+,+\s*$", "", val)
```

```
            ctx = normalize_spaces(t[max(0, m.start()-160): m.end()+160])
            key = ("auditor", val)
            if val and key not in seen:
                facts.append({"type": "auditor", "value": val, "context": ctx})
                seen.add(key)

        # --- Headquarters / Registered Office (single-line capture) ---
        hq_pat = (
            r"\b("
            r"Registered\s+Office|Headquarters?|Corporate\s+Office"
            r")\b[:\-—]?\s*([^\n]{10,200})"
        )
        for m in re.finditer(hq_pat, t, flags=re.I):
            val = normalize_spaces(m.group(2))
            # remove trailing "Tel/Phone/Email" if present on same line
            val = re.split(r"\b(Tel\.?|Phone|Email|Fax)\b", val, maxsplit=1)[0].strip(" ,;.-")
            key = ("hq", val)
            if val and key not in seen:
                facts.append({"type": "hq", "value": val, "context": val})
                seen.add(key)

        # --- Business segments (list after keyword; split on common separators) ---
        seg_match = re.search(
            r"\b(business\s+segments?|reportable\s+segments?)\b[:\-—]?\s*([^\n]{10,300})",
            t, flags=re.I
        )
        if seg_match:
            raw = seg_match.group(2)
            # split on ; , / • | and 'and' when used with commas
            parts = re.split(r"[;,/•|]+|\s+\band\b\s+", raw, flags=re.I)
            segs = []
            for s in parts:
                s = s.strip(" ;,./•|-")
                # keep mid-length tokens; drop obvious noise
                if 1 < len(s) <= 80 and not re.fullmatch(r"\d+|page\s*\d+", s, flags=re.I):
                    segs.append(s)
            segs = sorted(set(segs), key=str.lower)
            if segs:
```

```
            val = ", ".join(segs)
            ctx = normalize_spaces(seg_match.group(0))
            key = ("segments", val)
            if key not in seen:
                facts.append({"type": "segments", "value": val, "context": ctx})
                seen.add(key)

    return facts
```

Build more questions from textual facts

```
_MULTI_WS_RE   = re.compile(r"\s+")
_END_PUNC_RE   = re.compile(r"[,\.;:]+$")
_YEAR_RE       = re.compile(r"\b(20\d{2})\b")
_FY_RE         = re.compile(r"\bfy\s*'?\s*(\d{2,4})\b", re.I)
_HONORIFICS_RE = re.compile(r"^(mr\.?|mrs\.?|ms\.?|dr\.?|shri|smt\.?)\s+", re.I)

def _norm_answer(s: str) -> str:
    s = s.strip()
    s = _HONORIFICS_RE.sub("", s)
    s = _MULTI_WS_RE.sub(" ", s)
    s = _END_PUNC_RE.sub("", s)
    return s

def _infer_year_from_ctx(ctx: str):
    y = _YEAR_RE.search(ctx)
    if y:
        yr = int(y.group(1))
        return yr if 2000 <= yr <= 2100 else None
    fy = _FY_RE.search(ctx)
    if fy:
        n = int(fy.group(1))
        if n < 100:
            n += 2000
        return n if 2000 <= n <= 2100 else None
```

```
        return None

    def _base_confidence(metric: str, ctx_len: int) -> float:
        # simple heuristic by metric type + context size
        base = {
            "ceo": 0.9, "cfo": 0.9, "auditor": 0.85,
            "hq": 0.8, "segments": 0.75
        }.get(metric, 0.8)
        if ctx_len > 180: base += 0.03
        if ctx_len < 60:  base -= 0.05
        return max(0.6, min(0.98, base))

    def textual_facts_to_qas(doc_name: str, company: str, text: str) -> List[Dict]:
        out = []
        facts = extract_textual_facts(text)
        seen = set()  # (metric, normalized_answer)

        for f in facts:
            t = f.get("type"); v = f.get("value", ""); ctx = f.get("context", "")
            if not t or not v:
                continue

            # Map fact type → question
            if t == "ceo":
                q = f"Who was the CEO of {company}?"
            elif t == "cfo":
                q = f"Who was the CFO of {company}?"
            elif t == "auditor":
                q = f"Who is the statutory auditor of {company}?"
            elif t == "hq":
                q = f"What is the registered office address of {company}?"
            elif t == "segments":
                q = f"What are the reportable business segments of {company}?"
            else:
                continue

            ans = _norm_answer(v)
            if not ans:
```

```
                continue

            key = (t, ans.lower())
            if key in seen:
                continue
            seen.add(key)

            yr = _infer_year_from_ctx(ctx)
            conf = _base_confidence(t, len(ctx))

            out.append({
                "question": q,
                "answer": ans,
                "metric": t,
                "year": yr,                     # keep None if not inferred (same field name)
                "company": company,
                "source_doc": doc_name,
                "context_snippet": ctx,
                "confidence_heuristic": conf
            })

        return out
```

Generate more YoY comparisons (for many metrics)

```
_NUM_TOKEN = re.compile(r"^\(?-?\s*(?:\d{1,3}(?:[,\s]\d{3})*|\d+)(?:\.\d+)?\)?$")
_EXTRACT_NUM = re.compile(r"-?\d+(?:[\.,]\d+)?")
_HAS_CURR_OR_UNIT = re.compile(r"(₹|Rs\.?|INR|USD|EUR|\$|€|£|crore|cr\.?|cr|lakh|million|mn|billion|bn|thousand|k)\

def _to_int_year(y) -> int:
    try:
        return int(y)
    except Exception:
        return None
```

```
    def _clean_ans_str(s: str) -> str:
        return (s or "").strip()

    def _parse_amount_to_float(s: str):
        """
        Very light parser: strips currency, commas; handles parentheses as negatives; ignores units scaling.
        Returns float or None. (We do NOT rescale crore/million to a base—just numeric core for delta%)
        """
        if not s:
            return None
        s = s.strip()
        neg = False
        if s.startswith("(") and s.endswith(")"):
            neg = True
        # keep first numeric span
        m = _EXTRACT_NUM.search(s.replace(",", "").replace(" ", ""))
        if not m:
            return None
        try:
            val = float(m.group(0).replace(",", "").replace(" ", ""))
            return -val if neg else val
        except Exception:
            return None

    def _answer_quality_key(ans: str, conf: float):
        """
        Ranking key for choosing the best answer among duplicates in a year.
        Prefer presence of currency/unit, then length, then confidence.
        """
        ans = ans or ""
        has_cu = 1 if _HAS_CURR_OR_UNIT.search(ans) else 0
        return (has_cu, len(ans), conf if isinstance(conf, (int, float)) else 0.0)

    def build_yoy_pairs_multi(items: List[Dict[str, Any]], metrics: List[str], company: str) -> List[Dict]:
        out = []
        for metric in metrics:
            # collect by year with multiple candidates
            by_year = {}
```

```
conf_by_year = {}

for it in items:
    if it.get("metric") != metric:
        continue
    if it.get("company") != company:
        continue
    y = _to_int_year(it.get("year"))
    if y is None:
        continue
    ans = _clean_ans_str(it.get("answer", ""))
    if not ans or ans == "1":
        continue

    conf = it.get("confidence_heuristic", 0.7)
    # keep best candidate per year by quality key
    prev = by_year.get(y)
    if prev is None:
        by_year[y] = ans
        conf_by_year[y] = conf
    else:
        # choose better
        cur_key  = _answer_quality_key(ans, conf)
        prev_key = _answer_quality_key(prev, conf_by_year.get(y, 0.0))
        if cur_key > prev_key:
            by_year[y] = ans
            conf_by_year[y] = conf

years_sorted = sorted(by_year.keys(), reverse=True)
# need at least two distinct years
if len(years_sorted) < 2:
    continue

y1, y2 = years_sorted[0], years_sorted[1]  # most recent, previous
a1, a2 = by_year[y1], by_year[y2]

# compose question / answer
q = f"Compare {company}'s {metric.replace('_',' ')} in {y2} vs {y1}."
```

```
            ans_str = f"{y2}: {a2}; {y1}: {a1}."

            # try approximate YoY delta if both parse
            v1 = _parse_amount_to_float(a1)
            v2 = _parse_amount_to_float(a2)
            yoy_note = ""
            if v1 is not None and v2 is not None and v2 != 0:
                try:
                    pct = ((v1 - v2) / abs(v2)) * 100.0
                    yoy_note = f" (~{pct:+.1f}% YoY)"
                except Exception:
                    pass

            if yoy_note:
                ans_str = ans_str[:-1] + yoy_note + "."

            # confidence: average of two years with small boost if delta computed
            base_conf = 0.72
            c1 = conf_by_year.get(y1, base_conf)
            c2 = conf_by_year.get(y2, base_conf)
            conf_out = max(0.6, min(0.95, (c1 + c2) / 2 + (0.03 if yoy_note else 0.0)))

            out.append({
                "question": q,
                "answer": ans_str,
                "metric": f"{metric}_comparison",
                "year": f"{y2} vs {y1}",
                "company": company,
                "source_doc": "multiple",
                "context_snippet": "",
                "confidence_heuristic": conf_out
            })
        return out
```

Build Q/A Candidates

```
# Optional explicit mapping beats heuristic (edit as needed)
COMPANY_MAP = {
    "infosys-ar-25": "Infosys",
    "annual-report-2024": "Infosys",
}

def company_for_doc(doc_name: str, full_text: str) -> str:
    return COMPANY_MAP.get(doc_name, guess_company_from_text(doc_name, full_text))

# Metrics that are % by nature
PCT_METRICS = {"operating_margin", "net_margin"}
PCT_ONLY_RE = re.compile(r"^\s*\d+(?:\.\d+)?\s*%\s*$", re.I)
HAS_CURRENCY_OR_UNIT = re.compile(r"(₹|Rs\.?|INR|USD|EUR|\$|€|£|crore|cr\.?|cr|lakh|million|mn|billion|bn|thousand|
HAS_NUMBER = re.compile(r"\d")

def _clean_answer(s: str) -> str:
    return normalize_spaces((s or "").replace(" ,", ",")).strip()

def _compose_question(company: str, metric: str, year):
    base = {
        "revenue":             "revenue",
        "net_income":          "net income",
        "ebitda":              "EBITDA",
        "eps":                 "EPS",
        "cash_flow":           "operating cash flow",
        "assets":              "total assets",
        "liabilities":         "total liabilities",
        "operating_margin":    "operating margin",
        "net_margin":          "net margin",
        "headcount":           "total employee headcount",
        "dividend":            "dividend",
        "equity":              "total equity",
        "cash_and_cash_eq":    "cash and cash equivalents",
        "rd_expense":          "R&D expense",
        "capex":               "capital expenditure (CapEx)",
        "opex":                "operating expenses (OpEx)",
    }.get(metric, metric.replace("_", " "))
```

```
        if year is not None:
            return f"What was {company}'s {base} in {year}?"
        else:
            return f"What was {company}'s {base}?"

    def _confidence(metric: str, amount: str, year, ctx: str) -> float:
        # start higher if we have year + numeric signal
        c = 0.65
        if year is not None:
            c += 0.15
        if HAS_NUMBER.search(amount or ""):
            c += 0.1
        if HAS_CURRENCY_OR_UNIT.search(amount or ""):
            c += 0.05
        if len(ctx or "") > 140:
            c += 0.02
        # percentage metrics: boost if % present
        if metric in PCT_METRICS and PCT_ONLY_RE.search(amount or ""):
            c += 0.05
        return max(0.6, min(0.98, c))

    # ---- Rebuild candidates with extra metrics + textual facts (same outputs) ----
    texts = load_clean_texts(CLEAN_DIR)
    sections = load_sections(CLEAN_DIR)

    candidates = []
    seen = set()  # (question, answer) to dedupe

    for doc_name, txt in sorted(texts.items()):
        company = company_for_doc(doc_name, txt)

        # numeric/amount-like facts
        for f in extract_all_metrics(txt):
            metric, amount, year, ctx = f.get("metric"), f.get("amount"), f.get("year"), f.get("context", "")
            if not metric:
                continue
            amount = _clean_answer(amount)
```

```
            # Skip junk placeholders and empty values
            if not amount or amount == "1":
                continue

            # For non-% metrics, avoid pure percentages like "12%"
            if metric not in PCT_METRICS and PCT_ONLY_RE.fullmatch(amount):
                continue

            q = _compose_question(company, metric, year)
            conf = _confidence(metric, amount, year, ctx)

            row = {
                "question": q,
                "answer": amount,
                "metric": metric,
                "year": year,
                "company": company,
                "source_doc": doc_name,
                "context_snippet": ctx,
                "confidence_heuristic": conf
            }
            key = (row["question"], row["answer"])
            if key not in seen:
                seen.add(key)
                candidates.append(row)

        # textual facts (CEO/CFO/auditor/HQ/segments)
        for row in textual_facts_to_qas(doc_name, company, txt):
            key = (row["question"], row["answer"])
            if key not in seen:
                seen.add(key)
                candidates.append(row)

    print("Candidates so far:", len(candidates))
```

```
Candidates so far: 188
```

Deduplicate & Add YoY Comparison Q/As

```
import random
from collections import defaultdict

# ---------------- De-duplicate with highest-confidence winner ----------------
def _norm(s):
    return normalize_spaces((s or "")).lower()

best = {}
for it in candidates:
    key = (_norm(it.get("question")), _norm(it.get("answer")))
    prev = best.get(key)
    if (prev is None) or (it.get("confidence_heuristic", 0) > prev.get("confidence_heuristic", 0)):
        best[key] = it

uniq = list(best.values())

# ---------------- Collect companies & build YoY pairs (de-duplicated) ----------------
companies = sorted({it["company"] for it in uniq})
metrics_for_yoy = ["revenue","net_income","ebitda","eps","operating_margin","net_margin","cash_flow"]

yoy_more_raw = []
for comp in companies:
    yoy_more_raw += build_yoy_pairs_multi(uniq, metrics_for_yoy, comp)

# de-dupe YoY QAs too (same criterion)
best_yoy = {}
for it in yoy_more_raw:
    key = (_norm(it.get("question")), _norm(it.get("answer")))
    prev = best_yoy.get(key)
    if (prev is None) or (it.get("confidence_heuristic", 0) > prev.get("confidence_heuristic", 0)):
        best_yoy[key] = it
yoy_more = list(best_yoy.values())

# ---------------- Controls: ambiguous + irrelevant (unchanged schema) ----------------
```

```
controls = [
    {"question": "What was the revenue?", "answer": "Not in scope", "metric":"control_ambiguous", "year": None, "co
    {"question": "Tell me about future mergers?", "answer": "Not in scope", "metric":"control_irrelevant", "year":
    {"question": "What is the capital of France?", "answer": "Not in scope", "metric":"control_irrelevant", "year":
]

qa_dataset = uniq + yoy_more + controls
print("Total after YoY+controls:", len(qa_dataset))

# ---------------- Balanced sample to ~50 with diversity + stability ----------------
def balanced_sample(items: List[Dict[str, Any]], max_total: int = 50, seed: int = 42):
    """
    Stratify into numeric / textual / comparison / control buckets, then:
      - keep top by confidence
      - enforce diversity across (company, metric)
      - stable randomness (seeded) within ties
    Output shape unchanged: returns a list of dict rows.
    """
    random.seed(seed)

    # Bucketize
    buckets = {"numeric": [], "textual": [], "comparison": [], "control": []}
    for it in items:
        m = it.get("metric", "")
        if m.endswith("_comparison"):
            buckets["comparison"].append(it)
        elif m.startswith("control_"):
            buckets["control"].append(it)
        elif m in {"ceo","cfo","auditor","hq","segments"}:
            buckets["textual"].append(it)
        else:
            buckets["numeric"].append(it)

    # Diversity helper: pick greedily across (company, metric) to avoid clumping
    def pick_diverse(rows, k):
        rows = sorted(rows, key=lambda x: x.get("confidence_heuristic", 0), reverse=True)
        chosen, seen_pair = [], set()
        # first pass — strict company+metric diversity
```

```
            for r in rows:
                pair = (r.get("company"), r.get("metric"))
                if pair not in seen_pair:
                    chosen.append(r)
                    seen_pair.add(pair)
                    if len(chosen) >= k:
                        return chosen
            # second pass — relax to company-only diversity
            seen_co = set([c.get("company") for c in chosen])
            for r in rows:
                if r in chosen:
                    continue
                co = r.get("company")
                if co not in seen_co:
                    chosen.append(r); seen_co.add(co)
                    if len(chosen) >= k:
                        return chosen
            # final pass — fill remaining by confidence
            for r in rows:
                if r not in chosen:
                    chosen.append(r)
                    if len(chosen) >= k:
                        return chosen
            return chosen

        # Targets per bucket (same totals as before)
        target_numeric     = 22
        target_textual     = 10
        target_comparison  = 12
        target_control     = min(6, len(buckets["control"]))

        take = []
        take += pick_diverse(buckets["numeric"],    target_numeric)
        take += pick_diverse(buckets["textual"],    target_textual)
        take += pick_diverse(buckets["comparison"], target_comparison)

        # Controls: take as-is (already small)
        take += sorted(buckets["control"], key=lambda x: x.get("confidence_heuristic",0), reverse=True)[:target_contro
```

```
    # If we overshoot, trim by lowest confidence but keep at least 1 per (company,metric) if possible
    if len(take) > max_total:
        # tag with rank to prefer keeping diverse items first
        def score(item):
            base = item.get("confidence_heuristic", 0)
            # bonus for diversity
            bonus = 0.02
            return base + bonus
        take = sorted(take, key=score, reverse=True)[:max_total]

    # Finally, ensure deterministic order (by bucket preference + confidence)
    def bucket_rank(it):
        m = it.get("metric", "")
        if m.endswith("_comparison"): return 1
        if m.startswith("control_"):  return 4
        if m in {"ceo","cfo","auditor","hq","segments"}: return 2
        return 0  # numeric default

    take = sorted(take, key=lambda x: (bucket_rank(x), -x.get("confidence_heuristic",0)))
    return take

qa_balanced = balanced_sample(qa_dataset, max_total=50)
print("Balanced size:", len(qa_balanced))

# Quick peek
pd.DataFrame(qa_balanced).head(10)
```

```
Total after YoY+controls: 193
Balanced size: 37
```

	question	answer	metric	year	company	source_doc	context_snippet	confidence_heuristic
0	What was Infosys's net income in 2023?	23,268	net_income	2023	Infosys	annual-report-2024	at April 1, 2022 2,103 2,844 55,449 7,926 (264...	0.92
1	What was Infosys's operating margin in 2020?	20.7	operating_margin	2020	Infosys	annual-report-2024	May 31, 2024 Dinesh R. Co-Head of Delivery Bus...	0.92
2	What was Infosys's total employee headcount in...	1,882	headcount	2024	Infosys	annual-report-2024	Italy Malta Denmark Finland Sweden Hong Kong T...	0.92
3	What was Infosys's dividend in 2024?	(2)	dividend	2024	Infosys	annual-report-2024	unil Kumar Dhareshwar Global Head – Corporate ...	0.92
4	What was Infosys's total equity in 2024?	(1)	equity	2024	Infosys	annual-report-2024	dian equity shares: INE009A01021 Dematerializa...	0.92
	What was Infosys's					annual-report-	at April 1, 2022 2,103 2,844	

Balance to ~50 Q/As

```
import random
from collections import defaultdict
from typing import List, Dict, Any

def balanced_sample(items: List[Dict[str, Any]], per_metric: int = 8, max_total: int = 50):
    """
    Returns a diversified subset:
      - up to `per_metric` items per metric (round-robin across companies)
      - overall capped at `max_total`
    Input/Output schema unchanged.
    """
    # group by metric
```

```
        by_metric: Dict[str, List[Dict[str, Any]]] = defaultdict(list)
        for it in items:
            by_metric[it.get("metric", "unknown")].append(it)

        def _year_int(y):
            try:
                return int(y)
            except Exception:
                return -10**9  # non-year sorts last

        # de-dup helper within the builder
        def _dedupe(rows):
            seen = set()
            out = []
            for r in rows:
                key = (r.get("question", "").strip().lower(), r.get("answer", "").strip().lower())
                if key in seen:
                    continue
                seen.add(key)
                out.append(r)
            return out

        sampled = []

        for metric, rows in by_metric.items():
            if not rows:
                continue

            # group by company for round-robin
            by_co: Dict[str, List[Dict[str, Any]]] = defaultdict(list)
            for r in rows:
                by_co[r.get("company", "N/A")].append(r)

            # sort each company's rows by (confidence desc, year desc)
            for co in by_co:
                by_co[co].sort(
                    key=lambda x: (x.get("confidence_heuristic", 0.0), _year_int(x.get("year"))),
                    reverse=True
```

```
                )

            # round-robin pick to per_metric
            picked, ptr = [], {co: 0 for co in by_co}
            companies = sorted(by_co.keys())  # deterministic order
            while len(picked) < per_metric:
                advanced = False
                for co in companies:
                    i = ptr[co]
                    if i < len(by_co[co]):
                        picked.append(by_co[co][i])
                        ptr[co] = i + 1
                        advanced = True
                        if len(picked) >= per_metric:
                            break
                if not advanced:  # all exhausted
                    break

            # if still short (few companies), fill from remaining pool by (conf, year)
            if len(picked) < per_metric:
                pool = []
                for co, arr in by_co.items():
                    pool.extend(arr[ptr[co]:])
                pool.sort(key=lambda x: (x.get("confidence_heuristic", 0.0), _year_int(x.get("year"))), reverse=True)
                for r in pool:
                    if len(picked) >= per_metric:
                        break
                    picked.append(r)

            sampled.extend(_dedupe(picked))

        # global de-dup
        sampled = _dedupe(sampled)

        # if more than max_total, trim but try to keep at least 1 per metric
        if len(sampled) > max_total:
            # ensure at least one per metric first
            keep_min = []
```

```
        seen_metric = set()
        for metric, rows in by_metric.items():
            # pick the best for this metric that exists in sampled
            candidates = [r for r in sampled if r.get("metric") == metric]
            if candidates:
                best = max(
                    candidates,
                    key=lambda x: (x.get("confidence_heuristic", 0.0), _year_int(x.get("year")))
                )
                keep_min.append(best)
                seen_metric.add(metric)

        # remaining pool without the ones we've locked in
        locked_ids = set(id(x) for x in keep_min)
        remaining = [r for r in sampled if id(r) not in locked_ids]

        # sort remaining by (confidence desc, year desc)
        remaining.sort(key=lambda x: (x.get("confidence_heuristic", 0.0), _year_int(x.get("year"))), reverse=True)

        # fill up to max_total
        target = max_total - len(keep_min)
        sampled = keep_min + remaining[:max(0, target)]

    return sampled
```

```
qa_balanced = balanced_sample(qa_dataset, per_metric=8, max_total=50)
print("Total candidates:", len(qa_dataset))
print("Balanced sample:", len(qa_balanced))
df_preview = pd.DataFrame(qa_balanced)[["question","answer","metric","year","company","source_doc","confidence_heu
df_preview.head(15)
# df_preview
```

```
Total candidates: 193
Balanced sample: 50
```

	question	answer	metric	year	company	source_doc	confidence_heuristic
0	What was Infosys's revenue?	89,032	revenue	None	Infosys	annual-report-2024	0.77
1	What was Infosys's net income in 2025?	26,713	net_income	2025	Infosys	infosys-ar-25	0.92
2	What was Infosys's operating cash flow?	29,022	cash_flow	None	Infosys	annual-report-2024	0.77
3	What was Infosys's total assets?	26.6	assets	None	Infosys	annual-report-2024	0.77
4	What was Infosys's operating margin in 2025?	21.1	operating_margin	2025	Infosys	infosys-ar-25	0.92
5	What was Infosys's total employee headcount in...	1,869	headcount	2025	Infosys	infosys-ar-25	0.92
6	What was Infosys's dividend in 2024?	(2)	dividend	2024	Infosys	annual-report-2024	0.92
7	What was Infosys's total equity in 2024?	(1)	equity	2024	Infosys	annual-report-2024	0.92
8	What was Infosys's cash and cash equivalents?	2.9	cash_and_cash_eq	None	Infosys	annual-report-2024	0.77
9	What was Infosys's capital	(1)	capex	None	Infosys	annual-report-	0.77

Save Q/As (JSONL + CSV)

```
to_save = qa_balanced if len(qa_balanced) >= 40 else qa_dataset  # fall back if not enough
OUT_QA_JSONL.parent.mkdir(parents=True, exist_ok=True)

with open(OUT_QA_JSONL, "w", encoding="utf-8") as f:
```

```
        for row in to_save:
            f.write(json.dumps(row, ensure_ascii=False) + "\n")

pd.DataFrame(to_save).to_csv(OUT_QA_CSV, index=False)

print("Saved:")
print(" -", OUT_QA_JSONL.resolve())
print(" -", OUT_QA_CSV.resolve())
print("Total Q/A pairs saved:", len(to_save))
```

```
Saved:
 - /content/drive/MyDrive/RAG-FT-DATA/qa_pairs.jsonl
 - /content/drive/MyDrive/RAG-FT-DATA/qa_pairs.csv
Total Q/A pairs saved: 50
```

```
df = pd.read_csv(OUT_QA_CSV)
display_cols = ["question","answer","metric","year","company","source_doc","confidence_heuristic"]
df.sort_values(by="confidence_heuristic", ascending=False)[display_cols].head(50)
```

	question	answer	metric	year	company	source_doc	confidence_heuristic
16	Compare Infosys's net income in 2024 vs 2025.	2024: 26,233; 2025: 26,713 (~+1.8% YoY).	net_income_comparison	2024 vs 2025	Infosys	multiple	0.95
17	Compare Infosys's operating margin in 2020 vs ...	2020: 20.7; 2025: 21.1 (~+1.9% YoY).	operating_margin_comparison	2020 vs 2025	Infosys	multiple	0.95
12	Who was the CEO of Infosys?	and Managing Director Awards	ceo	2025	Infosys	infosys-ar-25	0.93
13	Who was the CFO of Infosys?	ESG Committee Refer to	cfo	2030	Infosys	annual-report-2024	0.93
33	Who was the CFO of Infosys?	of Infosys Limited	cfo	2024	Infosys	annual-report-2024	0.93
32	Who was the CFO of Infosys?	certification. Establishment of vigil	cfo	2024	Infosys	annual-report-2024	0.93

Step 2 (RAG)

```
# ------------------------------------------------------------------
# Improved, robust indexing bootstrap (same artifact names preserved)
# ------------------------------------------------------------------

# Reproducibility
SEED = 42
random.seed(SEED)
np.random.seed(SEED)
os.environ["PYTHONHASHSEED"] = str(SEED)

# Model choices
EMBED_MODEL_NAME = "sentence-transformers/all-MiniLM-L6-v2"   # 384-d

CHUNKS_PATH = ROOT / "processed"
BM25_DIR    = ROOT / "embeddings" / "bm25_index"
FAISS_DIR   = ROOT / "embeddings" / "faiss_index"


# File locations (reuse your dirs)
```

```
CHUNK_JSONL = CHUNKS_PATH / "chunks_400.jsonl"
BM25_PKL    = BM25_DIR / "bm25.pkl"
BM25_META   = BM25_DIR / "bm25_metadata.csv"

FAISS_IDX   = FAISS_DIR / "faiss.index"
ID_MAP_JSON = FAISS_DIR / "id_map.json"
EMB_NPY     = FAISS_DIR / "embeddings.npy"

def _normalize_ws(s: str) -> str:
    return re.sub(r"\s+", " ", (s or "").strip())

def _tokenize_bm25(s: str) -> list:
    # lowercase; keep alnum + currency/percent; split on others
    s = s.lower()
    s = re.sub(r"[^a-z0-9₹$£€%\.]+", " ", s)
    return [t for t in s.split() if t and len(t) > 1]
```

```
!pip install faiss-cpu
```

```
Collecting faiss-cpu
  Downloading faiss_cpu-1.12.0-cp312-cp312-manylinux_2_27_x86_64.manylinux_2_28_x86_64.whl.metadata (5.1 kB)
Requirement already satisfied: numpy<3.0,>=1.25.0 in /usr/local/lib/python3.12/dist-packages (from faiss-cpu) (2.0.2)
Requirement already satisfied: packaging in /usr/local/lib/python3.12/dist-packages (from faiss-cpu) (25.0)
Downloading faiss_cpu-1.12.0-cp312-cp312-manylinux_2_27_x86_64.manylinux_2_28_x86_64.whl (31.4 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 31.4/31.4 MB 30.4 MB/s eta 0:00:00
Installing collected packages: faiss-cpu
Successfully installed faiss-cpu-1.12.0
```

```
!pip install rank_bm25
```

```
Collecting rank_bm25
  Downloading rank_bm25-0.2.2-py3-none-any.whl.metadata (3.2 kB)
Requirement already satisfied: numpy in /usr/local/lib/python3.12/dist-packages (from rank_bm25) (2.0.2)
Downloading rank_bm25-0.2.2-py3-none-any.whl (8.6 kB)
Installing collected packages: rank_bm25
Successfully installed rank_bm25-0.2.2
```

36	total employee headcount in...	1,882	headcount	2024	Infosys	annual-report-2024	0.92
37	What was Infosys's net profit in 2024?	26,233	net profit	2024	Infosys	annual-report-2024	0.92
38	What was Infosys's net profit in 2024?	25,568	net profit	2024	Infosys	infosys-ar-25	0.92
39	What was Infosys's net income in 2023?	23,268	net_income	2023	Infosys	annual-report-2024	0.92
40	What was Infosys's net income in 2023?	27,234	net_income	2023	Infosys	annual-report-2024	0.92
41	What was Infosys's net income in 2023?	24,095	net_income	2023	Infosys	annual-report-2024	0.92
6	What was Infosys's	(2)	dividend	2024	Infosys	annual-report-	0.92

dividend in 2024? 2024

7 What was Infosys's total equity in (1) equity 2024 Infosys annual-report- 0.92

Load Chunks & Simple Preprocessing

```
def _load_chunks_jsonl(path: Path) -> Tuple[list, list, list]:
    """
    Returns: ids, texts, meta (dict per chunk)
    Ensures each chunk has: chunk_id, text, doc_name, pages_approx
    """
    ids, texts, meta = [], [], []
    with open(path, "r", encoding="utf-8") as f:
        for ln in f:
            ln = ln.strip()
            if not ln:
                continue
            try:
                rec = json.loads(ln)
            except Exception:
                continue
            cid = rec.get("chunk_id")
            txt = _normalize_ws(rec.get("text", ""))
            if not cid or not txt:
                continue
            ids.append(cid)
            texts.append(txt)
            meta.append({
                "doc_name": rec.get("doc_name"),
                "pages_approx": rec.get("pages_approx", []),
                "chunk_size": rec.get("chunk_size")
            })
    return ids, texts, meta

print("Using chunks file:", CHUNK_JSONL)
ids, texts, meta = _load_chunks_jsonl(CHUNK_JSONL)
assert ids, f"No valid chunks in {CHUNK_JSONL}"
```

```
Using chunks file: /content/drive/MyDrive/RAG-FT-DATA/processed/chunks_400.jsonl
```

```
# De-duplicate by chunk_id (keep first)
seen, keep = set(), []
for i, cid in enumerate(ids):
    if cid in seen:
        continue
    seen.add(cid)
    keep.append(i)
ids   = [ids[i] for i in keep]
texts = [texts[i] for i in keep]
meta  = [meta[i] for i in keep]
print(f"[LOAD] Chunks: {len(ids):,}")

# build the `chunks` list many downstream utils expect
chunks = [
    {
        "chunk_id": cid,
        "text": txt,
        "doc_name": m.get("doc_name"),
        "pages_approx": m.get("pages_approx", []),
        "chunk_size": m.get("chunk_size"),
    }
    for cid, txt, m in zip(ids, texts, meta)
]
```

```
[LOAD] Chunks: 1,576
```

Text Normalization & Tokenizer (for BM25)

```
# Lightweight stopword list (no external downloads)
STOPWORDS = set("""
a an the and or of to in for on at from by with about as is are was were be been being this that these those
it its itself they them their we us our he she his her you your i me my mine ours yours theirs
""".strip().split())

TOKEN_RE = re.compile(r"[A-Za-z0-9]+")  # alphanum tokens
```

```
def normalize_text(s: str) -> str:
    s = s.lower()
    s = re.sub(r"\s+", " ", s).strip()
    return s

def tokenize_for_bow(s: str) -> List[str]:
    s = normalize_text(s)
    toks = TOKEN_RE.findall(s)
    return [t for t in toks if t not in STOPWORDS and len(t) > 1]
```

Build BM25 (Sparse Index)

```
import re, pickle
import pandas as pd
from pathlib import Path
from rank_bm25 import BM25Okapi

def build_or_load_bm25(ids, texts):
    if BM25_PKL.exists() and BM25_META.exists():
        try:
            with open(BM25_PKL, "rb") as f:
                obj = pickle.load(f)
            bm25 = obj.get("bm25")
            if bm25 is not None:
                print(f"[BM25] Loaded cached: {BM25_PKL.name}")
                return bm25
        except Exception:
            pass
        print("[BM25] Cache invalid, rebuilding...")

    print("[BM25] Building index…")
    corpus_tokens = [_tokenize_bm25(t) for t in texts]
    bm25 = BM25Okapi(corpus_tokens)

    # Persist BM25 + metadata (same filenames you used)
```

```
        with open(BM25_PKL, "wb") as f:
            pickle.dump({"bm25": bm25}, f)
        meta_df = pd.DataFrame([{
            "chunk_id": ids[i],
            "doc_name": meta[i]["doc_name"],
            "pages_approx": meta[i]["pages_approx"]
        } for i in range(len(ids))])
        meta_df.to_csv(BM25_META, index=False)
        print(f"[BM25] Saved: {BM25_PKL.name}, {BM25_META.name}")
        return bm25

    bm25 = build_or_load_bm25(ids, texts)
    print("[BM25] Ready. Corpus size:", len(texts))
```

```
[BM25] Loaded cached: bm25.pkl
[BM25] Ready. Corpus size: 1576
```

Build Dense Embeddings + FAISS (Cosine Similarity)

```
from sentence_transformers import SentenceTransformer, CrossEncoder

# Load or build embeddings
def build_or_load_embeddings(texts):
    if EMB_NPY.exists():
        try:
            arr = np.load(EMB_NPY)
            if arr.shape[0] == len(texts):
                print(f"[EMB] Loaded cached: {EMB_NPY.name} {arr.shape}")
                return arr
            else:
                print("[EMB] Cache size mismatch, rebuilding…")
        except Exception:
            print("[EMB] Cache read failed, rebuilding…")

    print("[EMB] Encoding chunks…")
    embedder = SentenceTransformer(EMBED_MODEL_NAME)
```

```
        emb = embedder.encode(
            texts, batch_size=64, show_progress_bar=True,
            convert_to_numpy=True, normalize_embeddings=True
        ).astype("float32")
        np.save(EMB_NPY, emb)
        print(f"[EMB] Saved: {EMB_NPY.name} {emb.shape}")
        return emb

    emb = build_or_load_embeddings(texts)
    dim = emb.shape[1]
```

```
[EMB] Loaded cached: embeddings.npy (1576, 384)
```

```
import faiss

# Load or build FAISS index (cosine via IP on normalized vectors)
def build_or_load_faiss(emb):
    if FAISS_IDX.exists():
        try:
            idx = faiss.read_index(str(FAISS_IDX))
            if idx.ntotal == emb.shape[0]:
                print(f"[FAISS] Loaded cached: {FAISS_IDX.name} (ntotal={idx.ntotal})")
                return idx
            else:
                print("[FAISS] Cache size mismatch, rebuilding…")
        except Exception:
            print("[FAISS] Cache read failed, rebuilding…")

    print("[FAISS] Building index (IP on normalized vectors = cosine)…")
    idx = faiss.IndexFlatIP(dim)
    idx.add(emb)
    faiss.write_index(idx, str(FAISS_IDX))
    print(f"[FAISS] Saved: {FAISS_IDX.name} (ntotal={idx.ntotal})")
    return idx

index = build_or_load_faiss(emb)
```

```
# Save / refresh id_map (same schema/filename you used)
id_map = {
    "chunk_ids": ids,
    "doc_names": [m["doc_name"] for m in meta],
    "pages_approx": [m["pages_approx"] for m in meta],
}
with open(ID_MAP_JSON, "w", encoding="utf-8") as f:
    json.dump(id_map, f, ensure_ascii=False, indent=2)

print("FAISS index ready. Vectors:", emb.shape[0], "dim:", dim)

# Quick stats
print(f"[STATS] chunks={len(ids):,} | dim={dim} | BM25 OK | FAISS ntotal={index.ntotal}")
```

```
[FAISS] Loaded cached: faiss.index (ntotal=1576)
FAISS index ready. Vectors: 1576 dim: 384
[STATS] chunks=1,576 | dim=384 | BM25 OK | FAISS ntotal=1576
```

Retrieval Helpers (Dense, Sparse, Fusion)

```
# ---------------------------- loaders ----------------------------
def load_faiss_and_map():
    """
    Load FAISS index + id_map with sanity checks.
    Returns (index, id_map) exactly like before.
    """
    try:
        idx = faiss.read_index(str(FAISS_DIR / "faiss.index"))
    except Exception as e:
        raise RuntimeError(f"Failed to read FAISS index: {e}")

    try:
        with open(FAISS_DIR / "id_map.json", "r", encoding="utf-8") as f:
            id_map = json.load(f)
    except Exception as e:
        raise RuntimeError(f"Failed to read id_map.json: {e}")
```

```
    # light sanity checks (won't change the return type)
    if not isinstance(id_map, dict) or "chunk_ids" not in id_map:
        raise RuntimeError("id_map.json missing required keys (chunk_ids, doc_names, pages_approx).")
    if idx.ntotal != len(id_map["chunk_ids"]):
        # still return; just warn so downstream can rebuild if needed
        print(f"[WARN] FAISS ntotal ({idx.ntotal}) != id_map size ({len(id_map['chunk_ids'])}).")

    return idx, id_map

# ---------------------------- helpers ----------------------------
_Q_WS = re.compile(r"\s+")
def _normalize_query(q: str) -> str:
    return _Q_WS.sub(" ", (q or "").strip())

def _maybe_e5_format(q: str) -> str:
    # If you ever switch to E5 models, this keeps your API stable.
    # For all-MiniLM-L6-v2 this is a no-op.
    if "intfloat/e5" in (EMBED_MODEL_NAME or ""):
        return f"query: {q}"
    return q

# ---------------------------- dense search ----------------------------
def dense_search(query: str, top_k: int = 10) -> List[Tuple[int, float]]:
    """Returns list of (row_index_in_chunks, score) for top_k."""
    q = _normalize_query(query)
    q = _maybe_e5_format(q)

    # guard: top_k cannot exceed index size
    k = min(max(int(top_k or 0), 1), index.ntotal if hasattr(index, "ntotal") else top_k)

    # encode (SentenceTransformer already on CPU/GPU as configured earlier)
    embedder = SentenceTransformer(EMBED_MODEL_NAME)
    q_emb = embedder.encode([q], convert_to_numpy=True, normalize_embeddings=True).astype("float32")
    D, I = index.search(q_emb, k)  # inner product scores (cosine if normalized)

    # FAISS can return -1 for empty rows in some scenarios; filter them
    hits = []
```

```
        for idx_i, score in zip(I[0].tolist(), D[0].tolist()):
            if idx_i is None or idx_i < 0:
                continue
            hits.append((int(idx_i), float(score)))
        return hits

    # -------------------------- sparse (BM25) --------------------------
    def sparse_search(query: str, top_k: int = 10) -> List[Tuple[int, float]]:
        """Returns list of (row_index_in_chunks, score) for top_k (BM25)."""
        toks = tokenize_for_bow(_normalize_query(query))
        if not toks:
            return []
        scores = bm25.get_scores(toks)
        if scores is None or len(scores) == 0:
            return []
        k = min(max(int(top_k or 0), 1), len(scores))
        idxs = np.argpartition(scores, -k)[-k:]  # faster than full sort
        # order by score desc
        idxs = idxs[np.argsort(scores[idxs])[::-1]]
        return [(int(i), float(scores[i])) for i in idxs]

    # -------------------------- fusion --------------------------
    def reciprocal_rank_fusion(dense_res, sparse_res, k: int = 60, top_k: int = 20):
        """
        Combine rankings via Reciprocal Rank Fusion (robust to scale differences).
        Inputs: lists of (idx, score) sorted by descending score.
        Output: list[(idx, fused_score)] (desc).
        """
        if not dense_res and not sparse_res:
            return []

        # record ranks
        ranks = defaultdict(lambda: {"dense": None, "sparse": None})
        for r, (i, _) in enumerate(dense_res, start=1):
            ranks[i]["dense"] = r
        for r, (i, _) in enumerate(sparse_res, start=1):
            # do not overwrite if already seen with a better (smaller) rank
            if ranks[i]["sparse"] is None or r < ranks[i]["sparse"]:
```

```
            ranks[i]["sparse"] = r

    fused = []
    K = float(k if k and k > 0 else 60)
    for i, rs in ranks.items():
        r_dense  = rs["dense"]  if rs["dense"]  is not None else 10**9
        r_sparse = rs["sparse"] if rs["sparse"] is not None else 10**9
        score = (1.0 / (K + r_dense)) + (1.0 / (K + r_sparse))
        fused.append((i, score))

    fused.sort(key=lambda x: x[1], reverse=True)
    # cap to available chunk count
    cap = min(int(top_k or 20), len(fused))
    return fused[:cap]

# ---------------------------- pretty ----------------------------
def pretty_hit(row_idx: int, score: float) -> Dict:
    """
    Returns a dict with chunk metadata and a short preview.
    If row_idx is invalid, returns a minimal placeholder dict.
    """
    try:
        c = chunks[row_idx]
        txt = c.get("text", "")
        return {
            "chunk_id":      c.get("chunk_id"),
            "doc_name":      c.get("doc_name"),
            "pages_approx":  c.get("pages_approx"),
            "score":         round(float(score), 4) if score is not None else None,
            "preview":       (txt[:220] + " ...") if isinstance(txt, str) and len(txt) > 220 else txt
        }
    except Exception:
        return {
            "chunk_id": None,
            "doc_name": None,
            "pages_approx": None,
            "score": round(float(score), 4) if score is not None else None,
```

```
            "preview": ""
        }
```

Cross-Encoder Re-Ranking (Advanced RAG)

```
# Cross-encoder for re-ranking (query, passage) pairs
CROSS_ENCODER_NAME = "cross-encoder/ms-marco-MiniLM-L-6-v2"
cross_encoder = CrossEncoder(CROSS_ENCODER_NAME)  # device auto-chosen

# Heuristics for speed/safety
_CE_MAX_PASSAGE_CHARS = 1200   # ~ keeps within CE limits without heavy tokenization
_CE_BATCH = 64                 # adjust if you have more/less VRAM

def _ce_safe_text(txt: str) -> str:
    if not isinstance(txt, str):
        return ""
    txt = re.sub(r"\s+", " ", txt).strip()
    if len(txt) > _CE_MAX_PASSAGE_CHARS:
        txt = txt[:_CE_MAX_PASSAGE_CHARS]
    return txt

def rerank_with_cross_encoder(query: str, fused_hits: List[Tuple[int, float]], top_k: int = 5):
    """
    Re-rank fused hits using a cross-encoder. Returns top_k in new order with CE scores.
    """
    if not fused_hits:
        return []

    # De-duplicate indices while preserving order
    seen = set()
    uniq_hits = []
    for i, s in fused_hits:
        if i is None or i in seen or i < 0 or i >= len(chunks):
            continue
        seen.add(i)
        uniq_hits.append((i, s))
```

```
        if not uniq_hits:
            return []

        # Build (query, passage) pairs with safe truncation
        pairs = [(query, _ce_safe_text(chunks[i].get("text", ""))) for i, _ in uniq_hits]

        try:
            # Batched predict to avoid OOM on long lists
            scores = []
            for b in range(0, len(pairs), _CE_BATCH):
                batch = pairs[b:b+_CE_BATCH]
                scores.extend(cross_encoder.predict(batch).tolist())
            # Combine and sort by CE score desc
            reranked = [(uniq_hits[j][0], float(scores[j])) for j in range(len(uniq_hits))]
            reranked.sort(key=lambda x: x[1], reverse=True)
            return reranked[:min(top_k, len(reranked))]
        except Exception as e:
            # Fallback: return original fused order (use fused score)
            # Keeps the same return type.
            print(f"[WARN] Cross-encoder rerank failed: {e}. Falling back to fused order.")
            return uniq_hits[:min(top_k, len(uniq_hits))]
```

End-to-End: Hybrid Retrieval + Re-Ranking (Demo)

```
def _norm_query(q: str) -> str:
    return re.sub(r"\s+", " ", (q or "").strip())

def _looks_numeric(q: str) -> bool:
    return bool(re.search(r"(\d|₹|\$|INR|USD|EUR|%|FY\s*'?\d{2,4})", q, flags=re.I))

def _adaptive_ks(q: str, corpus_size: int,
                 k_dense=15, k_sparse=15, k_fused=20, k_final=5):
    # scale a bit with corpus size
    scale = 1.0 if corpus_size < 5_000 else 1.5 if corpus_size < 20_000 else 2.0
    is_num = _looks_numeric(q)
    # bias: numeric → more dense; fuzzy text → more sparse
```

```
        kd = int(round(k_dense * scale * (1.2 if is_num else 0.9)))
        ks = int(round(k_sparse * scale * (0.9 if is_num else 1.2)))
        kf = int(round(k_fused * scale))
        kf = max(kf, max(kd, ks))  # fused can't be smaller than sources
        kf = min(kf, corpus_size)  # cap by corpus
        kf_final = min(int(k_final), kf)
        return kd, ks, kf, kf_final

    def hybrid_retrieve(query: str, k_dense=15, k_sparse=15, k_fused=20, k_final=5):
        q = _norm_query(query)
        t0 = time.time()

        # adapt ks to query + corpus size (uses global `chunks`)
        corpus_size = len(chunks)
        kd, ks, kf, kfin = _adaptive_ks(q, corpus_size, k_dense, k_sparse, k_fused, k_final)

        # stage timings
        t_dense0 = time.time()
        d_hits = dense_search(q, top_k=kd) if kd > 0 else []
        t_dense1 = time.time()

        t_sparse0 = time.time()
        s_hits = sparse_search(q, top_k=ks) if ks > 0 else []
        t_sparse1 = time.time()

        # defensive: if one side is empty, still proceed with the other
        t_fuse0 = time.time()
        fused  = reciprocal_rank_fusion(d_hits, s_hits, k=60, top_k=kf) if (d_hits or s_hits) else []
        t_fuse1 = time.time()

        # budgeted + de-duplicated list for CE: keep top-8 from dense, top-8 from sparse, remainder from fused
        # (does not change outputs—only affects CE inputs)
        seen = set()
        ce_pool = []
        for lst in (d_hits[:8], s_hits[:8], fused):
            for i, sc in lst:
                if i not in seen and 0 <= i < len(chunks):
                    seen.add(i)
```

```
                ce_pool.append((i, sc))
            if len(ce_pool) >= max(kf, 32):  # keep CE batch small
                break
        if len(ce_pool) >= max(kf, 32):
            break

    t_rer0 = time.time()
    rerank = rerank_with_cross_encoder(q, ce_pool, top_k=kfin) if ce_pool else []
    t_rer1 = time.time()

    # fallbacks: if CE returns nothing, show fused; if fused empty, show dense; else sparse
    final_list = rerank if rerank else (fused[:kfin] if fused else (d_hits[:kfin] if d_hits else s_hits[:kfin]))

    out = {
        "query": query,
        "dense_top": [pretty_hit(i, sc) for i, sc in d_hits[:5]],
        "sparse_top": [pretty_hit(i, sc) for i, sc in s_hits[:5]],
        "fused_top": [pretty_hit(i, sc) for i, sc in fused[:5]],
        "reranked_top": [pretty_hit(i, sc) for i, sc in final_list],
        "latency_sec": round(time.time() - t0, 3)
    }
    # optional: print timing breakdown for tuning (remove if noisy)
    print(f"[TIMINGS] dense={t_dense1-t_dense0:.3f}s sparse={t_sparse1-t_sparse0:.3f}s "
          f"fuse={t_fuse1-t_fuse0:.3f}s rerank={t_rer1-t_rer0:.3f}s total={out['latency_sec']:.3f}s")
    return out


# Try a few queries (adjust to your reports)
queries = [
    "What was the company's revenue in 2025?",
    "What is the net profit for the year 2024?",
    "What were the total assets last year?",
]

for q in queries:
    out = hybrid_retrieve(q, k_dense=15, k_sparse=15, k_fused=20, k_final=5)
    print("\n=== QUERY:", q)
    print("Latency:", out["latency_sec"], "s")
```

```
    print("Top (re-ranked):")
    for h in out["reranked_top"]:
        print(f"  • {h['doc_name']} {h['pages_approx']} | score={h['score']}\n    {h['preview']}\n")
```

```
[TIMINGS] dense=1.269s sparse=0.005s fuse=0.000s rerank=5.943s total=7.218s

=== QUERY: What was the company's revenue in 2025?
Latency: 7.218 s
Top (re-ranked):
  • infosys-ar-25 [73, 74] | score=5.6627
    fiscal 2025 is ₹1,62,990 crore, a growth of 6.1%. Our revenues for fiscal 2025 in constant currency grew by 4.

  • infosys-ar-25 [242] | score=4.8223
    life on a systematic basis consistent with the transfer of goods or services to customer to which the asset re

  • annual-report-2024 [71] | score=3.1338
    currency terms, which represents the real growth in revenue, excluding the impact of currency fluctuations. We

  • infosys-ar-25 [228] | score=3.0643
    respectively and unbilled revenue amounting to ₹11,988 crore and ₹10,814 crore as at March 31, 2025 and March

  • infosys-ar-25 [242] | score=2.5076
    achievement of contractual milestones. The Company’s receivables are rights to consideration that are uncondit

[TIMINGS] dense=0.898s sparse=0.006s fuse=0.000s rerank=4.885s total=5.788s

=== QUERY: What is the net profit for the year 2024?
Latency: 5.788 s
Top (re-ranked):
  • infosys-ar-25 [238, 239] | score=5.0548
    Act, 1961. (Refer to Special Economic Zone Re-investment reserve under Note 2.12 Equity). Deferred income tax

  • annual-report-2024 [312, 313] | score=4.7383
    the extent its US branch's net profit during the year is greater than the increase in the net assets of the US

  • annual-report-2024 [73, 74] | score=4.1574
    equivalents and investments, excluding investments in equity, preference shares, compulsorily convertible debe

  • annual-report-2024 [235, 236] | score=4.0263
    such reserve by the Company for acquiring new plant and machinery for the purpose of its business as per the p
```

```
  • infosys-ar-25 [73, 74] | score=3.1025
    fiscal 2025 is ₹1,62,990 crore, a growth of 6.1%. Our revenues for fiscal 2025 in constant currency grew by 4.

[TIMINGS] dense=0.945s sparse=0.005s fuse=0.000s rerank=5.654s total=6.604s

=== QUERY: What were the total assets last year?
Latency: 6.604 s
Top (re-ranked):
  • infosys-ar-25 [208, 209] | score=-0.5082
    income from subsidiary in current year is ₹75 crore and in last year, it was ₹78 crore. The changes in the car

  • infosys-ar-25 [274, 275, 276] | score=-1.832
    2.10 11,940 12,808 Total current assets 97,099 89,432 Total assets 1,48,903 1,37,814 Consolidated Balance Shee

  • annual-report-2024 [270, 271, 272] | score=-2.0894
    1,245 Income tax assets (net) 2.17 3,045 6,453 Other non-current assets 2.10 2,121 2,318 Total non-current ass

  • annual-report-2024 [189, 190, 191, 192] | score=-2.4588
    30 days from the end of the financial year has not elapsed till the date of our report. For DELOITTE HASKINS &
```

Load a Small, Open-Source Generator (FLAN-T5)

```
# If needed:
# !pip install transformers accelerate sentencepiece bitsandbytes

import torch
from transformers import AutoTokenizer, AutoModelForSeq2SeqLM, GenerationConfig

# Choose your generator here:
# GEN_MODEL_NAME = "google/flan-t5-base"   # current
GEN_MODEL_NAME = "google/flan-t5-large"     # easy quality bump, if VRAM allows

device = "cuda" if torch.cuda.is_available() else "cpu"

# Try to pick an efficient dtype automatically
def _best_dtype():
```

```
        if device == "cuda":
            # bf16 on newer GPUs, else fp16
            return torch.bfloat16 if torch.cuda.is_bf16_supported() else torch.float16
        return torch.float32  # CPU

    # Optional: load 8-bit if bitsandbytes is present & on GPU
    USE_8BIT = (device == "cuda")
    load_kwargs = dict(
        torch_dtype=_best_dtype(),
        device_map="auto" if device == "cuda" else None,
        low_cpu_mem_usage=True
    )
    if USE_8BIT:
        try:
            import bitsandbytes as bnb  # noqa: F401
            load_kwargs.update(dict(load_in_8bit=True))
        except Exception:
            pass  # fall back to regular load

    gen_tokenizer = AutoTokenizer.from_pretrained(GEN_MODEL_NAME, use_fast=True)
    gen_model = AutoModelForSeq2SeqLM.from_pretrained(GEN_MODEL_NAME, **load_kwargs)

    print(f"Loaded generator: {GEN_MODEL_NAME} | device: {device} | dtype: {gen_model.dtype}")

    # Safer generation defaults (avoid 512-token overflow on T5):
    def make_gen_config(
        max_new_tokens=96,
        temperature=0.0,
        top_p=1.0,
        num_beams=1,
        repetition_penalty=1.0
    ):
        return GenerationConfig(
            max_new_tokens=max_new_tokens,
            do_sample=(num_beams == 1 and temperature > 0.0),
            temperature=temperature,
            top_p=top_p,
            num_beams=num_beams,
```

```
            repetition_penalty=repetition_penalty,
            pad_token_id=gen_tokenizer.pad_token_id,
            eos_token_id=gen_tokenizer.eos_token_id,
        )

    # Helper to encode safely within model's input window
    def encode_clipped(prompt: str):
        # Many T5 variants have ~512 token encoder limit
        max_in = getattr(gen_model.config, "max_position_embeddings", 512) or 512
        toks = gen_tokenizer(
            prompt,
            return_tensors="pt",
            truncation=True,
            max_length=max_in,
            padding=False
        )
        return {k: v.to(gen_model.device) for k, v in toks.items()}

    # Example usage:
    # cfg = make_gen_config(max_new_tokens=96, temperature=0.0, num_beams=1)
    # inputs = encode_clipped(prompt_text)
    # out_ids = gen_model.generate(**inputs, generation_config=cfg)
    # answer = gen_tokenizer.decode(out_ids[0], skip_special_tokens=True)
```

```
Loaded generator: google/flan-t5-large | device: cpu | dtype: torch.float32
```

Utilities: Trim Context to Fit the Model

```
# ---------- Token-budgeted concatenation (drop-in) ----------

def _model_ctx_limit(tokenizer, model, hard_fallback=512):
    # Try tokenizer first, then model config; fallback to provided hard cap
    for attr in ("model_max_length",):
        v = getattr(tokenizer, attr, None)
        if isinstance(v, int) and v > 0 and v < 10**9:  # avoid 'int(1e30)' sentinels
```

```
                return v
        for attr in ("n_positions", "max_position_embeddings"):
            v = getattr(getattr(model, "config", object()), attr, None)
            if isinstance(v, int) and v > 0:
                return v
        return hard_fallback

    def trim_context_to_tokens(texts, tokenizer, max_tokens: int = 768):
        """
        Concatenate passages and trim to a max token length for the generator input.
        Returns (joined_text, token_count).
        Optimized: pre-tokenizes each passage, accounts for separator cost once.
        """
        if not texts:
            return "", 0

        sep = "\n\n"
        sep_ids = tokenizer(sep, add_special_tokens=False, return_tensors="pt").input_ids
        sep_cost = sep_ids.shape[-1]

        # Pre-tokenize once per passage (no specials)
        toks_list = [
            tokenizer(t, add_special_tokens=False, return_tensors="pt").input_ids
            for t in texts
        ]
        joined_parts = []
        used = 0
        for i, ids in enumerate(toks_list):
            add_cost = ids.shape[-1] + (sep_cost if joined_parts else 0)
            if used + add_cost <= max_tokens:
                joined_parts.append(texts[i])
                used += add_cost
            else:
                break

        joined = sep.join(joined_parts)
        return joined, used
```

Compose Prompt + Generate Answer

```
# ---------- Prompt builder (fits to budget without repeated full tokenization) ----------

MAX_INPUT_TOKENS_HARD = 512  # safe default for T5-like models

def _build_prompt(query: str, passages: List[str]) -> str:
    context_block = "\n\n".join(f"- {p}" for p in passages) if passages else "(no context)"
    return (
        "You are a financial assistant. Answer the question using ONLY the given context. "
        "If the answer is not present, say 'Not in scope'. "
        "Be concise and report numbers exactly as in the context.\n\n"
        f"Context:\n{context_block}\n\n"
        f"Question: {query}\nAnswer:"
    )

def _fit_prompt_to_budget(query: str, passages: List[str], tokenizer, max_tokens: int) -> str:
    """
    Greedily add passages while guaranteeing room for the instruction + question.
    Uses pre-tokenized passage lengths and separator costs.
    """
    # Token cost for instruction + empty context + question
    base = _build_prompt(query, [])
    base_ids = tokenizer(base, return_tensors="pt", add_special_tokens=True, truncation=False).input_ids
    reserve = base_ids.shape[-1]

    if reserve >= max_tokens:
        # If even the header+question alone hits the limit, fall back to a minimal prompt
        minimal = f"Context:\n\nQuestion: {query}\nAnswer:"
        return minimal

    # Compute remaining budget for the context list
    budget = max_tokens - reserve

    # Pre-tokenize passages once (no specials), track sep cost for bullet+newlines
    bullet = "- "
```

```
        sep = "\n\n"
        bullet_cost = tokenizer(bullet, add_special_tokens=False, return_tensors="pt").input_ids.shape[-1]
        sep_cost = tokenizer(sep, add_special_tokens=False, return_tensors="pt").input_ids.shape[-1]

        kept = []
        used = 0
        for p in passages:
            # bullet + passage tokens; add sep if not first
            p_ids = tokenizer(p, add_special_tokens=False, return_tensors="pt").input_ids
            add_cost = bullet_cost + p_ids.shape[-1] + (sep_cost if kept else 0)
            if used + add_cost <= budget:
                kept.append(p)
                used += add_cost
            else:
                break

        return _build_prompt(query, kept)


    # ---------- Generation (safe defaults, device-aware) ----------

    @torch.no_grad()
    def generate_answer(query: str,
                        passages: List[str],
                        max_input_tokens: int = MAX_INPUT_TOKENS_HARD,
                        max_new_tokens: int = 64,
                        temperature: float = 0.0,
                        top_p: float = 1.0,
                        num_beams: int = 1) -> str:
        """
        Builds a prompt that fits within `max_input_tokens`, then generates.
        Only passes sampling kwargs when sampling is enabled.
        """
        # Respect model/tokenizer limits if available
        ctx_limit = _model_ctx_limit(gen_tokenizer, gen_model, hard_fallback=MAX_INPUT_TOKENS_HARD)
        max_input_tokens = min(int(max_input_tokens), ctx_limit)

        prompt = _fit_prompt_to_budget(query, passages, gen_tokenizer, max_tokens=max_input_tokens)
```

```
    enc = gen_tokenizer(
        prompt,
        return_tensors="pt",
        truncation=True,
        max_length=max_input_tokens,
        padding=False
    )
    enc = {k: v.to(gen_model.device) for k, v in enc.items()}

    do_sample = (temperature > 0.0 and num_beams == 1)
    gen_kwargs = dict(
        max_new_tokens=int(max_new_tokens),
        num_beams=int(num_beams),
        do_sample=bool(do_sample),
        pad_token_id=getattr(gen_tokenizer, "pad_token_id", None),
        eos_token_id=getattr(gen_tokenizer, "eos_token_id", None),
    )
    if do_sample:
        gen_kwargs.update(dict(temperature=float(temperature), top_p=float(top_p)))

    out = gen_model.generate(**enc, **gen_kwargs)
    ans = gen_tokenizer.decode(out[0], skip_special_tokens=True).strip()
    # light cleanup: collapse spaces
    ans = re.sub(r"\s+", " ", ans)
    return ans
```

Guardrails (Input + Output)

We'll implement two simple guardrails:

Input-side domain filter → if the question is clearly not finance/report related, we short-circuit with "Out of scope".

Output-side factuality check → if the generated answer contains a number not present in the retrieved context, flag as a possible hallucination (and optionally replace with "Not in scope").

```
import re
from typing import List, Dict, Any, Tuple

# ---------- Finance intent detection (expanded + name aligned) ----------

_FINANCE_KEYWORDS = [
    # statements & metrics
    r"revenue", r"net\s*sales", r"sales", r"turnover",
    r"net\s*(?:income|profit)", r"(?:operating|op\.?)\s*(?:income|profit)",
    r"ebitda", r"\bebit\b", r"\beps\b",
    r"cash(?:\s*and\s*cash\s*equivalents)?", r"operating\s*cash\s*flow", r"free\s*cash\s*flow",
    r"assets?", r"liabilit(?:y|ies)", r"net\s*worth", r"equity", r"debt",
    r"margin", r"gross\s*margin", r"net\s*margin", r"operating\s*margin",
    r"\bpbt\b", r"\bpat\b", r"dividend", r"capex", r"opex",
    r"receivables", r"payables", r"working\s*capital",
    r"guidance", r"outlook", r"\bmd&?a\b", r"management\s+discussion",
    r"segment(s)?", r"report", r"annual\s*report",
    r"balance\s*sheet", r"income\s*statement", r"statement\s*of\s*operations",
    r"cash\s*flow\s*statement",
    # governance / corp actions
    r"\bceo\b", r"\bcfo\b", r"chief\s+financial\s+officer", r"auditor",
    r"headcount", r"employee[s]?",
    r"\bmerger(s)?\b", r"\bacquisition(s)?\b", r"\bm&?a\b",
    # anchors
    r"\byoy\b", r"\bqoq\b", r"\bfx\b", r"\bforex\b"
]
FINANCE_RE = re.compile(r"\b(?:" + "|".join(_FINANCE_KEYWORDS) + r")\b", re.IGNORECASE)

# FY24 / FY2024 / FY 2023-24
FY_RE = re.compile(r"\bfy\s*'?(\d{2,4})(?:\s*[-/]\s*(\d{2,4}))?\b", re.IGNORECASE)

# Hard out-of-scope patterns (keep small & precise)
_HARD_OOS = re.compile(r"\b(capital of|weather|recipe|lyrics?|translate|distance to)\b", re.IGNORECASE)

def _fy_to_years(m) -> List[int]:
    a, b = m.group(1), m.group(2)
    def norm(y): y = int(y); return y + 2000 if y < 100 else y
    return sorted({norm(a), norm(b)}) if b else [norm(a)]
```

```
def is_financial_query(query: str, min_hits: int = 1) -> bool:
    """Return True for finance/corporate queries; False only for clearly irrelevant ones."""
    q = (query or "").lower()
    if _HARD_OOS.search(q):
        return False
    hits = len(FINANCE_RE.findall(q))
    fy_hits = len({tuple(m.groups(default="")) for m in FY_RE.finditer(q)})
    return (hits + fy_hits) >= min_hits


# ---------- 2) Number extraction (currency/units, accounting negatives) ----------

# Currency symbols & units
_CCY   = r"(?:₹|rs\.?|inr|usd|\$|eur|€|gbp|£)"
_UNITS = r"(?:percent|%|crore|cr\.?|cr|lakh|lakhs|million|mn|billion|bn|thousand|k)"

# Numbers incl. negatives, (accounting), decimals, grouped
_NUM_CORE = r"(?:\(?-?\d{1,3}(?:,\d{3})+|\(?-?\d+(?:\.\d+)?\)?)"

# Currency can appear before or after; unit typically after
NUM_WITH_META_RE = re.compile(
    rf"(?:{_CCY}\s*)?{_NUM_CORE}(?:\s*{_UNITS})?(?:\s*{_CCY})?",
    flags=re.IGNORECASE
)

# Simple numeric core (for verbatim/strict checks)
NUM_STRICT_RE = re.compile(r"\b-?(?:\d{1,3}(?:,\d{3})+|\d+(?:\.\d+)?)\b")

# Years to ignore in hallucination checks unless specifically asked
YEAR_RE = re.compile(r"\b(19|20)\d{2}\b")

def _strip_commas_parens(x: str) -> str:
    x = x.replace(",", "")
    if x.startswith("(") and x.endswith(")"):
        x = "-" + x[1:-1]
    return x
```

```
def extract_numbers_with_meta(s: str) -> List[Tuple[str, str]]:
    """
    Returns list of (value_str, unit_str) pairs.
    value_str is normalized (commas removed, accounting () → -).
    unit_str is one of %, crore, million, bn, etc. (lowercased); '' if none.
    """
    out = []
    if not s:
        return out
    for m in NUM_WITH_META_RE.finditer(s):
        span = m.group(0)
        # pull out unit (last occurrence)
        unit = ""
        um = re.search(_UNITS, span, flags=re.IGNORECASE)
        if um:
            unit = um.group(0).lower().strip().rstrip(".")
            if unit == "percent":
                unit = "%"
        # pull out numeric core
        nm = NUM_STRICT_RE.search(span)
        if not nm:
            continue
        val = _strip_commas_parens(nm.group(0))
        out.append((val, unit))
    return out

def extract_numbers_strict(s: str) -> List[str]:
    if not s:
        return []
    return [_strip_commas_parens(x) for x in NUM_STRICT_RE.findall(s)]

def _looks_like_year(x: str) -> bool:
    return bool(YEAR_RE.fullmatch(x))


# ---------- 3) Factuality / consistency check (unit-aware) ----------

# Map units to scale factors (for rough numeric equivalence)
```

```
    _UNIT_SCALE = {
        "%": 1.0,
        "million": 1_000_000.0, "mn": 1_000_000.0,
        "billion": 1_000_000_000.0, "bn": 1_000_000_000.0,
        "crore": 10_000_000.0, "cr": 10_000_000.0, "cr.": 10_000_000.0,
        "lakh": 100_000.0, "lakhs": 100_000.0,
        "thousand": 1_000.0, "k": 1_000.0,
        "": 1.0
    }

    def _unit_scale(u: str) -> float:
        return _UNIT_SCALE.get(u.lower(), 1.0)

    def _float_or_none(x: str):
        try:
            return float(x)
        except Exception:
            return None

    def _units_equivalent(a_unit: str, b_unit: str) -> bool:
        canon = {
            "%": {"%", "percent"},
            "million": {"mn", "million"},
            "billion": {"bn", "billion"},
            "crore": {"cr", "cr.", "crore"},
            "thousand": {"k", "thousand"},
            "": {""}
        }
        def norm(u: str) -> str:
            u = (u or "").lower().strip(".")
            for k, vs in canon.items():
                if u in vs:
                    return k
            return u
        return norm(a_unit) == norm(b_unit)

    def _to_base(val_str: str, unit: str) -> float:
        """Scale numeric string by unit into a comparable base value."""
```

```
        v = _float_or_none(val_str)
        if v is None:
            return None
        return v * _unit_scale(unit)

    def _roughly_equal_scaled(a_val: str, a_unit: str, b_val: str, b_unit: str, tol_rel: float = 0.01) -> bool:
        """
        Compare numbers after scaling (1% tolerance by default).
        Special-case %: compare raw numbers and ignore scaling.
        """
        if (a_unit == "%" or b_unit == "%"):
            fa, fb = _float_or_none(a_val), _float_or_none(b_val)
            if fa is None or fb is None:
                return False
            if fb == 0:
                return abs(fa) < 1e-9
            return abs(fa - fb) / max(1e-9, abs(fb)) <= tol_rel

        A = _to_base(a_val, a_unit)
        B = _to_base(b_val, b_unit)
        if A is None or B is None:
            return False
        if B == 0:
            return abs(A) < 1e-9
        return abs(A - B) / max(1e-9, abs(B)) <= tol_rel

    def output_factuality_check(answer: str, contexts: List[str], query: str = "") -> Dict[str, Any]:
        """
        Two-tier check:
          1) strict: numeric strings in answer must appear verbatim in contexts
          2) lenient: (value, unit) matches after unit normalization & scaling
        Ignores year-like tokens unless the query mentions 'year' explicitly.
        """
        ql = (query or "").lower()

        # Extract numbers from answer
        ans_nums_strict = [x for x in extract_numbers_strict(answer) if not _looks_like_year(x) or "year" in ql]
        ans_pairs = [(v, u) for (v, u) in extract_numbers_with_meta(answer)
```

```
                if (not _looks_like_year(v)) or ("year" in ql)]

        # Extract numbers from contexts (concat for pool)
        ctx_all_text = " ".join(contexts or [])
        ctx_nums_strict = set(extract_numbers_strict(ctx_all_text))
        ctx_pairs = extract_numbers_with_meta(ctx_all_text)

        # 1) Strict: missing verbatim numbers
        strict_missing = sorted({x for x in ans_nums_strict if x not in ctx_nums_strict})

        # 2) Lenient: try to justify via unit-aware scaled comparison
        unjustified = []
        for (aval, aunit) in ans_pairs:
            # if present strictly, it's fine
            if aval in ctx_nums_strict:
                continue
            # otherwise, look for a close scaled match with same/compatible unit family
            supported = False
            for (cval, cunit) in ctx_pairs:
                if not _units_equivalent(aunit, cunit) and not (aunit == "" or cunit == ""):
                    # allow bare numbers to match if scaled equivalents land close
                    pass
                if _roughly_equal_scaled(aval, aunit, cval, cunit, tol_rel=0.01):
                    supported = True
                    break
            if not supported:
                unjustified.append((aval, aunit))

        # Suspicious if any strict missing that also could not be justified
        suspicious_values = [v for v in strict_missing if all(v != uv for (uv, _) in unjustified)]

        return {
            "answer_numbers_strict": sorted(ans_nums_strict),
            "context_numbers_strict": sorted(ctx_nums_strict),
            "unjustified_pairs": unjustified,           # list of (value, unit) not supported by context
            "suspicious_numbers": suspicious_values,    # verbatim numbers not found nor justified
            "is_potential_hallucination": len(suspicious_values) > 0 or len(unjustified) > 0
```

```
    }
```

End-to-End rag_answer (Hybrid Retrieval → Re-rank → Generate → Guardrails)

```
import time
import re

def _norm_query(q: str) -> str:
    return re.sub(r"\s+", " ", (q or "").strip())

def _looks_numeric(q: str) -> bool:
    return bool(re.search(r"(\d|₹|\$|INR|USD|EUR|%|FY\s*'?\d{2,4})", q, flags=re.I))

def _adaptive_ks(q: str, corpus_size: int,
                 k_dense=15, k_sparse=15, k_fused=20, k_final=5):
    # Light scaling with corpus size
    scale = 1.0 if corpus_size < 5_000 else 1.5 if corpus_size < 20_000 else 2.0
    is_num = _looks_numeric(q)
    # bias: numeric → more dense; fuzzy → more sparse
    kd = int(round(k_dense * scale * (1.2 if is_num else 0.9)))
    ks = int(round(k_sparse * scale * (0.9 if is_num else 1.2)))
    kf = int(round(k_fused * scale))
    # caps
    kd = max(1, min(kd, corpus_size))
    ks = max(1, min(ks, corpus_size))
    kf = max(max(kd, ks), min(kf, corpus_size))
    kfin = max(1, min(int(k_final), kf))
    return kd, ks, kf, kfin

def _normalize_conf_from_scores(scores):
    if not scores:
        return 0.4
    mn, mx = min(scores), max(scores)
    if mx == mn:
        return 0.5
```

```
        # normalize first score (best) to [0.5, 1.0]
        return float(0.5 + 0.5 * ((scores[0] - mn) / (mx - mn)))

    # Acceptable “finished” numeric answer formats (preserves commas, units, %)
    _NUM_OK = re.compile(
        r'^(?:₹|\$|usd|inr|rs\.?)?\s*\(?-?[\d,]+(?:\.\d+)?\)?(?:\s*(?:%|crore|cr\.?|cr|million|mn|billion|bn|lakh|lakhs
        re.IGNORECASE
    )

    # Raw numeric-span extractor (keeps currency, commas, %, indian units)
    _NUM_SPAN = re.compile(
        r'(?:₹|\$|usd|inr|rs\.?)?\s*\(?-?\d{1,3}(?:,\d{3})+|-?\d+(?:\.\d+)?\)?\s*(?:%|crore|cr\.?|cr|million|mn|billior
        re.IGNORECASE
    )

    def _extract_spans(text: str):
        return [(m.group(0).strip(), m.start()) for m in _NUM_SPAN.finditer(text or "")]

    def _best_numeric_from_context(query: str, contexts: list[str]) -> str | None:
        """Heuristic: pick the numeric span most likely answering the query."""
        ql = (query or "").lower()
        want_currency = any(k in ql for k in ["revenue","sales","turnover","profit","cash","equity"])
        want_pct      = "margin" in ql or "%" in ql
        yr = (re.search(r'\b(19|20)\d{2}\b', ql) or [None]) and re.search(r'\b(19|20)\d{2}\b', ql)
        yr = yr.group(0) if yr else None

        best = (None, -1.0)
        for ctx in contexts or []:
            txt = re.sub(r"\s+", " ", str(ctx)).strip()
            spans = _extract_spans(txt)
            if not spans:
                continue
            for span, pos in spans:
                score = 0.0
                # closeness to keywords
                for kw in ("revenue","sales","turnover","profit","net profit","margin","cash","equity"):
                    m = re.search(kw, txt, re.I)
                    if m:
```

```
                    score += max(0.0, 1.0 - abs(pos - m.start())/150.0)
            # unit preference
            if want_pct and "%" in span: score += 0.6
            if want_currency and re.search(r'(₹|\$|usd|inr|rs\.?)', span, re.I): score += 0.6
            # same-year proximity
            if yr and yr in txt: score += 0.4
            # specificity bonus
            score += min(0.5, len(re.sub(r"[^\d]", "", span))/10.0)
            if score > best[1]:
                best = (span, score)
    return best[0]

def rag_answer(query: str,
               k_dense: int = 15,
               k_sparse: int = 15,
               k_fused: int = 20,
               k_final: int = 5,
               max_input_tokens: int = MAX_INPUT_TOKENS_HARD,
               generator_max_new_tokens: int = 64) -> Dict[str, Any]:
    t0 = time.time()
    q = _norm_query(query)

    # 1) INPUT GUARDRAIL (domain filter)
    if not is_finance_query(q):
        return {
            "query": query,
            "method": "RAG",
            "answer": "Out of scope (non-financial query).",
            "confidence": 0.3,
            "retrieved_contexts": [],
            "latency_sec": round(time.time() - t0, 3),
            "guardrail_triggered": "input_out_of_scope"
        }

    # 2) RETRIEVAL + FUSION (+ budgeted CE pool) + RERANK
    corpus_size = len(chunks)
    kd, ks, kf, kfin = _adaptive_ks(q, corpus_size, k_dense, k_sparse, k_fused, k_final)
```

```
d_hits = dense_search(q, top_k=kd) if kd > 0 else []           # [(idx, score)]
s_hits = sparse_search(q, top_k=ks) if ks > 0 else []           # [(idx, score)]
fused  = reciprocal_rank_fusion(d_hits, s_hits, k=60, top_k=kf) if (d_hits or s_hits) else []

# Build a small, diverse pool for CE (fast):
# take a few top from dense & sparse, then fill from fused (dedup)
seen = set()
ce_pool = []
for lst, take in ((d_hits, 8), (s_hits, 8), (fused, kf)):
    for i, sc in lst[:take]:
        if i is None or i in seen or i < 0 or i >= corpus_size:
            continue
        seen.add(i)
        ce_pool.append((i, sc))
        if len(ce_pool) >= max(32, kf):  # cap CE pool
            break
    if len(ce_pool) >= max(32, kf):
        break

rerank = rerank_with_cross_encoder(q, ce_pool, top_k=kfin) if ce_pool else []
used_list = rerank if rerank else (fused[:kfin] if fused else (d_hits[:kfin] if d_hits else s_hits[:kfin]))

# Gather full texts for factuality check & generation
top_idxs = [i for i, _ in used_list if 0 <= i < corpus_size]
contexts = [chunks[i]["text"] for i in top_idxs] if top_idxs else []

# 3) GENERATION (token-budgeted)
gen_t0 = time.time()
if contexts:
  # try to discourage 1-token stubs like "₹1"
  answer = generate_answer(
    query=q,
    passages=contexts,
    max_input_tokens=max_input_tokens,
    max_new_tokens=generator_max_new_tokens,
    temperature=0.0,
    top_p=1.0,
    # if your generate_answer forwards **kwargs to model.generate,
```

```
    # these two will help a lot; if it ignores them, the fallback below still fixes outputs:
    # min_new_tokens=4,
    # length_penalty=1.2,
    num_beams=6,
    # no_repeat_ngram_size=3,
  )
  answer = (answer or "").strip()
else:
  answer = "Not in scope"
gen_t1 = time.time()

# Extractive fallback: if the model didn't emit a well-formed numeric answer,
# copy the most relevant numeric span directly from the retrieved contexts.
_fallback_used = False
fb = None  # <-- init to avoid UnboundLocalError

# Extractive fallback only if we have contexts and the model's answer
# doesn't look like a well-formed numeric
if contexts and not _NUM_OK.match(answer):
    fb = _best_numeric_from_context(q, contexts)

if fb is not None:
    answer = fb
    _fallback_used = True
    # (optional) adjust confidence if you want
    # conf = max(conf, 0.75)  # do this later where you compute conf


# 4) OUTPUT GUARDRAIL (factuality/number-consistency vs contexts)
fact_check = output_factuality_check(answer, contexts, query=q)
guardrail_flag = None
final_answer = answer

# Confidence: CE if available, else fused/dense normalized
if rerank:
    ce_scores = [sc for _, sc in rerank]
    conf = _normalize_conf_from_scores(ce_scores)
elif fused:
```

```
            fused_scores = [sc for _, sc in fused]
            conf = _normalize_conf_from_scores(fused_scores)
        elif d_hits:
            conf = _normalize_conf_from_scores([sc for _, sc in d_hits])
        else:
            conf = 0.4

        if fact_check.get("is_potential_hallucination", False):
            guardrail_flag = "output_potential_hallucination"
            final_answer = "Not in scope (insufficient supporting context)."
            conf = min(conf, 0.35)  # downgrade on intervention

        # 5) Pretty payload of contexts (previews shown to user)
        pretty_contexts = []
        for (i, sc) in (rerank if rerank else used_list):
            if not (0 <= i < corpus_size):
                continue
            c = chunks[i]
            txt = c.get("text", "")
            preview = txt if len(txt) <= 280 else (txt[:280] + " ...")
            pretty_contexts.append({
                "chunk_id": c.get("chunk_id"),
                "doc_name": c.get("doc_name"),
                "pages_approx": c.get("pages_approx"),
                "ce_score": round(float(sc), 4) if sc is not None else None,
                "preview": preview
            })

        t1 = time.time()
        return {
            "query": query,
            "method": "RAG",
            "answer": final_answer,
            "raw_answer": answer,
            "confidence": round(float(conf), 3),
            "retrieved_contexts": pretty_contexts,
            "latency_sec": round(t1 - t0, 3),
            "gen_time_sec": round(gen_t1 - gen_t0, 3),
```

```
        "guardrail_triggered": guardrail_flag,
        "fact_check": fact_check
    }
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
/tmp/ipython-input-2573735762.py in <cell line: 0>()
     85                k_fused: int = 20,
     86                k_final: int = 5,
---> 87                max_input_tokens: int = MAX_INPUT_TOKENS_HARD,
     88                generator_max_new_tokens: int = 64) -> Dict[str, Any]:
     89     t0 = time.time()

NameError: name 'MAX_INPUT_TOKENS_HARD' is not defined
```

Try a Few End-to-End Queries

```
tests = [
    "What was the company's revenue in fiscal 2025?",
    "What was the net profit in 2024?",
    "Capital of France?"
]
for q in tests:
    out = rag_answer(q, k_dense=15, k_sparse=15, k_fused=20, k_final=5, max_input_tokens=512)
    print("\n=== QUERY:", q)
    print("Answer:", out["answer"])
    print("Confidence:", out["confidence"], "| Total latency:", out["latency_sec"], "s")
    print("Top contexts:")
    for ctx in out["retrieved_contexts"][:2]:
        print(f"  - {ctx['doc_name']} {ctx['pages_approx']} | ce={ctx['ce_score']}")
        print("    ", ctx["preview"][:160], "...")
```

```
---------------------------------------------------------------------------
UnboundLocalError                         Traceback (most recent call last)
/tmp/ipython-input-945938846.py in <cell line: 0>()
      5 ]
      6 for q in tests:
----> 7     out = rag_answer(q, k_dense=15, k_sparse=15, k_fused=20, k_final=5, max_input_tokens=512)
      8     print("\n=== QUERY:", q)
      9     print("Answer:", out["answer"])

/tmp/ipython-input-3061223710.py in rag_answer(query, k_dense, k_sparse, k_fused, k_final, max_input_tokens,
generator_max_new_tokens)
    160     if contexts and not _NUM_OK.match(answer):
    161       fb = _best_numeric_from_context(q, contexts)
--> 162     if fb:
    163       answer = fb
    164       _fallback_used = True

UnboundLocalError: cannot access local variable 'fb' where it is not associated with a value
```

The generator is defaulting to “Not in scope” even when the contexts are relevant.

Confidence stays high because it’s computed from CE scores (good retrieval) while the answer is a fallback.

Below is a surgical set of improvements.

```
# ========================
# RAG: Retrieval → Rerank → Generate → Guardrails (Improved)
# ========================
import re, time, torch
from typing import List, Dict, Any

# ---------- Retrieval query boosting (year/FY & finance hints) ----------
_YEAR_RE = re.compile(r"\b(20\d{2})\b")
def _expand_retrieval_query(q: str) -> str:
    """
    Expand query with finance hints + FY variants of any detected year
```

```
        (e.g., 2024 -> FY2024, FY24). Keeps original q intact and adds soft hints.
        """
        ql = (q or "").lower().strip()
        years = _YEAR_RE.findall(ql)
        fy_tokens = []
        for y in years:
            yy = int(y) % 100
            fy_tokens.extend([f"FY{y}", f"FY {y}", f"FY{yy}", f"FY {yy}"])
        # light-touch finance hints (don't flood with too many keywords)
        boost = " revenue profit net income ebitda eps crore billion inr ₹ usd $ fy"
        add = (" " + " ".join(sorted(set(fy_tokens)))) if fy_tokens else ""
        return q + " " + boost + add

    def dense_search(query: str, top_k: int = 10):
        q = _expand_retrieval_query(query)
        k = max(1, min(int(top_k or 0), index.ntotal if hasattr(index, "ntotal") else top_k))
        embedder = SentenceTransformer(EMBED_MODEL_NAME)
        q_emb = embedder.encode([q], convert_to_numpy=True, normalize_embeddings=True).astype("float32")
        D, I = index.search(q_emb, k)
        hits = []
        for idx, sc in zip(I[0].tolist(), D[0].tolist()):
            if idx is None or idx < 0:
                continue
            hits.append((int(idx), float(sc)))
        return hits  # [(idx, score)]

    def sparse_search(query: str, top_k: int = 10):
        q = _expand_retrieval_query(query)
        toks = tokenize_for_bow(q)
        if not toks:
            return []
        scores = bm25.get_scores(toks)
        if scores is None or len(scores) == 0:
            return []
        k = max(1, min(int(top_k or 0), len(scores)))
        idxs = np.argpartition(scores, -k)[-k:]
        idxs = idxs[np.argsort(scores[idxs])[::-1]]
        return [(int(i), float(scores[i])) for i in idxs]
```

```
# ---------- Prompt builders (token-budgeted) ----------
MAX_INPUT_TOKENS_HARD = 512  # T5-like cap

def _build_prompt(query: str, passages: List[str]) -> str:
    context_block = "\n\n".join(f"- {p}" for p in passages) if passages else "(no context)"
    return (
        "You are a financial assistant. Use ONLY the context to answer. "
        "If the answer is truly unavailable, say 'Not in scope'. "
        "Report numbers exactly as written (keep currency and units).\n\n"
        f"Context:\n{context_block}\n\n"
        f"Question: {query}\nAnswer:"
    )

def _build_numeric_only_prompt(query: str, passages: List[str]) -> str:
    """Force numeric-only answers when appropriate (revenue, net profit, EPS, EBITDA)."""
    ctx = "\n\n".join(passages) if passages else "(no context)"
    # return (
    #     "You are a financial QA assistant. Use ONLY the context.\n"
    #     "Return ONLY the numeric answer with currency/unit (no words; only digits, commas, currency symbols, %, a
    #     "If the answer is not present, return exactly: Not in scope\n\n"
    #     f"Context:\n{ctx}\n\nQuestion: {query}\nAnswer:"
    # )
    return (
        "You are a financial QA assistant. Use ONLY the context.\n"
        "Return a short summary of the answer with currency/unit (no words; only digits, commas, currency symbols,
        "If the answer is not present, return exactly: Not in scope\n\n"
        f"Context:\n{ctx}\n\nQuestion: {query}\nAnswer:"
    )

def _fit_prompt_to_budget(prompt_builder, query: str, passages: List[str], tokenizer, max_tokens: int) -> str:
    """
    Greedily add passages until the full prompt would exceed max_tokens.
    Works with both builders (_build_prompt or _build_numeric_only_prompt).
    """
    kept = []
    for p in passages:
        candidate = prompt_builder(query, kept + [p])
```

```
            ids = tokenizer(candidate, return_tensors="pt", add_special_tokens=True, truncation=False).input_ids
            if ids.shape[-1] <= max_tokens:
                kept.append(p)
            else:
                break
        return prompt_builder(query, kept)

    # ---------- Minimal extractor (extract-first) ----------
    _FINANCE_KEY_TO_REGEX = {
        "revenue":    r"\b(revenue|income\s+from\s+operations|total\s+income|turnover)\b",
        "net profit": r"\b(net\s+profit|profit\s+after\s+tax|pat)\b",
        "net income": r"\b(net\s+income|profit\s+after\s+tax|pat)\b",
        "ebitda":     r"\b(ebitda)\b",
        "eps":        r"\b(eps|earnings\s+per\s+share)\b",
    }
    _KEY_RES = {k: re.compile(v, re.IGNORECASE) for k, v in _FINANCE_KEY_TO_REGEX.items()}

    # Reuse your improved numeric patterns
    _CCY   = r"(?:₹|rs\.?|inr|usd|\$|eur|€|gbp|£)"
    _UNITS = r"(?:percent|%|crore|cr\.?|cr|lakh|lakhs|million|mn|billion|bn|thousand|k)"
    _NUM_CORE = r"(?:\(?-?\d{1,3}(?:,\d{3})+|\(?-?\d+(?:\.\d+)?\)?)"
    AMOUNT_RE = re.compile(rf"(?:{_CCY}\s*)?{_NUM_CORE}(?:\s*{_UNITS})?(?:\s*{_CCY})?", re.I)

    _SENT_SPLIT = re.compile(r"(?<=[\.\!\?:;])\s+(?=[A-Z\(])")
    def _sentences(txt: str) -> List[str]:
        sents = _SENT_SPLIT.split(txt or "")
        return [s.strip() for s in sents if len((s or "").strip()) > 3]

    def _target_metric(query: str) -> str | None:
        ql = (query or "").lower()
        for k in _FINANCE_KEY_TO_REGEX:
            if k in ql:
                return k
        return None

    def _year_pref(query: str) -> List[int]:
        ys = [int(y) for y in _YEAR_RE.findall(query or "")]
        return sorted(set(ys), reverse=True)
```

```
def _extract_candidate_from_contexts(query: str, contexts: List[str]) -> str | None:
    metric = _target_metric(query)
    if metric is None:
        return None

    met_pat = _KEY_RES[metric]
    yrs = _year_pref(query)  # [2025, 2024, ...]
    year_pat = None
    if yrs:
        y = yrs[0]
        yy = y % 100
        year_pat = re.compile(rf"(FY\s*{y}|FY\s*{yy}|{y})", re.IGNORECASE)

    # detect month words to downweight “March 31” style dates
    MONTH_RE = re.compile(
        r"\b(jan(?:uary)?|feb(?:ruary)?|mar(?:ch)?|apr(?:il)?|may|jun(?:e)?|jul(?:y)?|"
        r"aug(?:ust)?|sep(?:t|tember)?|oct(?:ober)?|nov(?:ember)?|dec(?:ember)?)\b",
        re.IGNORECASE
    )

    MONEY_UNITS = {"crore", "cr", "cr.", "million", "mn", "billion", "bn", "lakh", "lakhs"}
    MONEY_METRICS = {"revenue", "net profit", "net income", "ebitda"}
    is_money_metric = metric in MONEY_METRICS

    def _num_core(val: str) -> float | None:
        s = val.replace(",", "")
        if s.startswith("(") and s.endswith(")"):
            s = "-" + s[1:-1]
        try:
            return float(s)
        except Exception:
            return None

    def _looks_like_date_number(am: str, sent: str) -> bool:
        # e.g., "March 31" or "31 March"
        return bool(re.fullmatch(r"-?\d{1,2}", am.strip()) and MONTH_RE.search(sent))
```

```
        def _has_currency(am: str) -> bool:
            return bool(re.search(_CCY, am, re.IGNORECASE))

        def _has_commas(am: str) -> bool:
            return "," in am

        def _is_year_str(am: str) -> bool:
            return any(am.strip() == str(y) for y in yrs) if yrs else False

        def _money_like(am: str, unit: str, val: float | None) -> bool:
            # “money-like” if it has currency, or money unit, or looks like a large figure (>= 10,000) or has comma-gr
            if _has_currency(am): return True
            if unit in MONEY_UNITS: return True
            if _has_commas(am): return True
            if val is not None and abs(val) >= 10_000: return True
            return False

        best = None  # (score, amount_str)

        for ctx in contexts or []:
            for sent in _sentences(ctx):
                if not met_pat.search(sent):
                    continue

                # base score: prefer year/FY mention if query asked for it
                base = 0.0
                year_hit = bool(year_pat and year_pat.search(sent))
                if yrs:
                    base += 1.6 if year_hit else -1.2  # stronger penalty if asked year not present

                # cross-encoder relevance
                try:
                    ce = cross_encoder.predict([(query, sent)])
                    base += float(ce[0]) if hasattr(ce, "__len__") else float(ce)
                except Exception:
                    pass

                # collect amounts in sentence
```

```
                amounts = AMOUNT_RE.findall(sent)
                if not amounts:
                    continue

                for am in amounts:
                    am = am.strip()
                    # skip if the number equals a target year (hard rule for money answers)
                    if is_money_metric and _is_year_str(am):
                        continue

                    unit = ""
                    um = re.search(_UNITS, am, re.IGNORECASE)
                    if um:
                        unit = um.group(0).lower().strip().rstrip(".")

                    # sentence-level score starts here
                    score = base

                    # penalize date-like small integers (e.g., 31)
                    if _looks_like_date_number(am, sent):
                        score -= 3.0

                    # numeric magnitude bonus (log-scale, capped)
                    val = _num_core(am)
                    if val is not None and val > 0:
                        import math
                        score += min(2.0, math.log10(val + 1.0))

                    # money-metric specific: require “money-like” or demote heavily
                    if is_money_metric:
                        if unit == "%":
                            score -= 2.5
                        if not _money_like(am, unit, val):
                            # Demote hard if it doesn’t look like a currency figure
                            score -= 3.0

                    # EPS: avoid %; allow small decimals
                    if metric == "eps":
```

```
                    if unit == "%":
                        score -= 1.5
                    if val is not None and val < 1000:
                        score += 0.5

                if (best is None) or (score > best[0]):
                    best = (score, am)

    return best[1] if best else None


def _first_money_like_from_contexts(passages: List[str]) -> str | None:
    for ctx in passages or []:
        # prefer currency/unit hits
        for m in AMOUNT_RE.finditer(ctx):
            am = m.group(0).strip()
            if re.search(_CCY, am, re.I) or re.search(_UNITS, am, re.I):
                return am
        # fallback: any amount with comma-grouping (large magnitude)
        for m in AMOUNT_RE.finditer(ctx):
            am = m.group(0).strip()
            if "," in am:
                return am
    return None

@torch.no_grad()
def generate_answer(query: str,
                    passages: List[str],
                    max_input_tokens: int = MAX_INPUT_TOKENS_HARD,
                    max_new_tokens: int = 64,
                    temperature: float = 0.0,
                    top_p: float = 1.0,
                    num_beams: int = 1) -> str:
    ql = (query or "").lower()
    numeric_intent = any(k in ql for k in ("revenue","net profit","net income","eps","ebitda"))
    builder = _build_numeric_only_prompt if numeric_intent else _build_prompt

    max_input_tokens = min(int(max_input_tokens), MAX_INPUT_TOKENS_HARD)
```

```
        prompt = _fit_prompt_to_budget(builder, query, passages, gen_tokenizer, max_tokens=max_input_tokens)

        enc = gen_tokenizer(prompt, return_tensors="pt", truncation=True, max_length=max_input_tokens, padding=False)
        enc = {k: v.to(gen_model.device) for k, v in enc.items()}

        do_sample = (temperature > 0.0 and num_beams == 1)
        gen_kwargs = dict(max_new_tokens=int(max_new_tokens), num_beams=int(num_beams), do_sample=bool(do_sample),
                          pad_token_id=getattr(gen_tokenizer, "pad_token_id", None),
                          eos_token_id=getattr(gen_tokenizer, "eos_token_id", None))
        if do_sample:
            gen_kwargs.update(dict(temperature=float(temperature), top_p=float(top_p)))

        out = gen_model.generate(**enc, **gen_kwargs)
        ans = gen_tokenizer.decode(out[0], skip_special_tokens=True).strip()

        if numeric_intent:
            # keep only money-like amount
            m = AMOUNT_RE.search(ans)
            if m:
                cand = m.group(0).strip()
                # reject if the “answer” is just a year (e.g., 2025)
                if not re.fullmatch(r"(19|20)\d{2}", cand):
                    return cand
            # fallback to contexts if the model emitted a bare year or no amount
            fallback = _first_money_like_from_contexts(passages)
            return fallback if fallback else "Not in scope"

        return ans

    # ---------- End-to-end RAG ----------
    def rag_answer(query: str,
                   k_dense: int = 15,
                   k_sparse: int = 15,
                   k_fused: int = 20,
                   k_final: int | None = None,
                   max_input_tokens: int = MAX_INPUT_TOKENS_HARD,
                   generator_max_new_tokens: int = 64) -> Dict[str, Any]:
        t0 = time.time()
```

```
        # Input guardrail (domain)
        if not is_finance_query(query):
            return {
                "query": query, "method": "RAG",
                "answer": "Out of scope (non-financial query).",
                "confidence": 0.3, "retrieved_contexts": [],
                "latency_sec": round(time.time() - t0, 3),
                "guardrail_triggered": "input_out_of_scope"
            }

        # Dynamic k_final: slightly higher for numeric factoids
        if k_final is None:
            k_final = 8 if any(w in (query or "").lower() for w in ("revenue","net profit","net income","eps","ebitda"

        # Retrieve → fuse → rerank
        d_hits = dense_search(query, top_k=k_dense)
        s_hits = sparse_search(query, top_k=k_sparse)
        fused  = reciprocal_rank_fusion(d_hits, s_hits, k=60, top_k=k_fused)
        rerank = rerank_with_cross_encoder(query, fused, top_k=k_final) if fused else []

        # Fallback if CE/rerank empty
        used_list = rerank if rerank else (fused[:k_final] if fused else (d_hits[:k_final] if d_hits else s_hits[:k_fi
        top_idxs = [i for i, _ in used_list]
        contexts = [chunks[i]["text"] for i in top_idxs] if top_idxs else []

        # --- Extract-first: prefer exact numeric from context when possible ---
        candidate = _extract_candidate_from_contexts(query, contexts)
        used_mode = "extractive" if candidate else "generative"

        # Generate (token-budgeted; numeric-only when needed)
        gen_t0 = time.time()
        answer = candidate if candidate else generate_answer(
            query, contexts,
            max_input_tokens=max_input_tokens,
            max_new_tokens=generator_max_new_tokens,
            temperature=0.0, top_p=1.0, num_beams=1
        )
```

```
gen_t1 = time.time()

# Confidence from CE/fused scores (min-max → [0.5, 1.0])
ce_scores = [sc for _, sc in rerank] if rerank else [sc for _, sc in used_list] if used_list else []
if ce_scores:
    mn, mx = min(ce_scores), max(ce_scores)
    conf = 0.5 if mx == mn else (ce_scores[0] - mn) / (mx - mn)
    conf = float(0.5 + 0.5 * conf)
else:
    conf = 0.4
if used_mode == "extractive":
    conf = max(conf, 0.85)  # boost extractive certainty

guardrail_flag = None
final_answer = answer

# If generator said Not in scope, keep extractor result if present
if used_mode == "generative" and final_answer.strip().lower().startswith("not in scope"):
    if candidate:
        final_answer = candidate
        guardrail_flag = "generator_conservative_auto_fill"
        conf = max(conf, 0.6)
    else:
        conf = min(conf, 0.35)

# Factuality check (numbers vs contexts)
fact_check = output_factuality_check(final_answer, contexts, query=query)
if fact_check.get("is_potential_hallucination", False):
    guardrail_flag = ("output_potential_hallucination" if not guardrail_flag
                      else guardrail_flag + "|output_potential_hallucination")
    final_answer = "Not in scope (insufficient supporting context)."
    conf = 0.35

# Pretty contexts (previews for UI)
pretty_contexts = []
for (i, sc) in (rerank if rerank else used_list):
    c = chunks[i]
    txt = c["text"]
```

```
            preview = txt if len(txt) <= 280 else (txt[:280] + " ...")
            pretty_contexts.append({
                "chunk_id": c["chunk_id"],
                "doc_name": c["doc_name"],
                "pages_approx": c["pages_approx"],
                "ce_score": round(float(sc), 4),
                "preview": preview
            })

        t1 = time.time()
        return {
            "query": query, "method": "RAG",
            "answer": final_answer, "raw_answer": answer,
            "confidence": round(conf, 3),
            "retrieved_contexts": pretty_contexts,
            "latency_sec": round(t1 - t0, 3),
            "gen_time_sec": round(gen_t1 - gen_t0, 3),
            "guardrail_triggered": guardrail_flag,
            "fact_check": fact_check
        }
```

```
tests = [
    "What was the company's revenue in fiscal 2025?",
    "What was the net profit in 2024?",
    "Capital of France?"
]
for q in tests:
    out = rag_answer(q, k_dense=15, k_sparse=15, k_fused=20, k_final=5, max_input_tokens=512)
    print("\n=== QUERY:", q)
    print("Answer:", out["answer"])
    print("Confidence:", out["confidence"], "| Total latency:", out["latency_sec"], "s")
    print("Top contexts:")
    for ctx in out["retrieved_contexts"][:2]:
        print(f"  - {ctx['doc_name']} {ctx['pages_approx']} | ce={ctx['ce_score']}")
        print("    ", ctx["preview"][:160], "...")
```

```
=== QUERY: What was the company's revenue in fiscal 2025?
Answer: ₹4,404 crore
Confidence: 1.0 | Total latency: 5.648 s
Top contexts:
  - infosys-ar-25 [73, 74] | ce=7.3158
     fiscal 2025 is ₹1,62,990 crore, a growth of 6.1%. Our revenues for fiscal 2025 in constant currency grew by 4.2
  - infosys-ar-25 [73, 74] | ce=4.3372
     Third-party items bought for service delivery to clients include software and hardware, which are integral to c

=== QUERY: What was the net profit in 2024?
Answer: 26,713
Confidence: 1.0 | Total latency: 9.178 s
Top contexts:
  - infosys-ar-25 [15, 16] | ce=3.2205
     Net profit*# 26,713 26,233 24,095 22,110 19,351 Basic earnings per share (in ₹)* 64.50 63.39 57.63 52.52 45.61
  - infosys-ar-25 [73, 74] | ce=1.848
     fiscal 2025 is ₹1,62,990 crore, a growth of 6.1%. Our revenues for fiscal 2025 in constant currency grew by 4.2

=== QUERY: Capital of France?
Answer: Out of scope (non-financial query).
Confidence: 0.3 | Total latency: 0.0 s
Top contexts:
```

Step 3 (Fine-Tuning)

We'll use FLAN-T5 (open-source, instruction-tuned) for a generative Q&A model, and we'll:

Load your Q/A dataset

Run a baseline (pre-fine-tuning) evaluation

Fine-tune FLAN-T5 with HuggingFace Trainer

Implement an advanced method: Adapter-based Mixture-of-Experts (AdapterFusion)

Provide an ft_answer() helper with a simple guardrail

Setup & Imports

```
# ========================
# Step 3 — Fine-Tuning: Setup & Configuration (improved)
# ========================
# If needed:
# !pip install -q transformers accelerate datasets sentencepiece evaluate scikit-learn

from pathlib import Path
import json, time, re, random, math, os
from typing import List, Dict, Any, Tuple

import numpy as np
import pandas as pd
import torch
from datasets import Dataset, DatasetDict, load_dataset
from transformers import (
    AutoTokenizer, AutoModelForSeq2SeqLM, DataCollatorForSeq2Seq,
    TrainingArguments, Trainer
)
import evaluate

QA_JSONL = ROOT / "qa_pairs.jsonl"
FT_OUT_M = Path(ROOT) / "fine_tuned_model"
FT_OUT   = FT_OUT_M / "ft_model"
FT_OUT.mkdir(parents=True, exist_ok=True)

# ---- Reproducibility ----
SEED = 42
random.seed(SEED)
np.random.seed(SEED)
torch.manual_seed(SEED)
if torch.cuda.is_available():
    torch.cuda.manual_seed_all(SEED)
torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmark = False
```

```
# ---- Device & mixed-precision helpers ----
device = "cuda" if torch.cuda.is_available() else "cpu"
def has_bf16() -> bool:
    # Many A100/H100/RTX 40xx support bf16; fallback to fp16 if not
    return torch.cuda.is_available() and torch.cuda.is_bf16_supported()

def mp_kwargs():
    if has_bf16():
        return dict(bf16=True, fp16=False)
    elif torch.cuda.is_available():
        return dict(bf16=False, fp16=True)
    else:
        return dict(bf16=False, fp16=False)

print(f"Using device: {device} | bf16={mp_kwargs().get('bf16', False)} fp16={mp_kwargs().get('fp16', False)}")

# ---- Configuration (edit here if needed) ----
CFG = {
    # Small seq2seq is fine for QA-style finetuning; keep same family as your generator
    "base_model": "google/flan-t5-base",   # or "google/flan-t5-small" if RAM-limited
    "max_input_len": 512,                  # budget for prompt+context if you include any
    "max_target_len": 64,                  # short answers for finance QA
    "train_batch_size": 8,                 # tune per GPU memory
    "eval_batch_size": 16,
    "grad_accum_steps": 2,                 # effective batch = 8*2 = 16
    "lr": 2e-4,
    "weight_decay": 0.01,
    "num_epochs": 5,
    "warmup_ratio": 0.06,
    "lr_scheduler": "cosine",              # linear/cosine/cosine_with_restarts
    "gradient_checkpointing": True,        # saves memory; slight compute overhead
    "logging_steps": 50,
    "eval_strategy": "steps",              # "epoch" or "steps"
    "eval_steps": 200,
    "save_steps": 200,
    "save_total_limit": 2,
    "early_stop_patience": 3,              # used with EarlyStoppingCallback (next cell)
```

```
    "report_to": "none",                    # or "tensorboard"
}

# ---- Data presence & quick sanity check ----
assert QA_JSONL.exists(), f"Missing {QA_JSONL}. Run Step 1 to create QA pairs."

# Peek at dataset size quickly
with open(QA_JSONL, "r", encoding="utf-8") as f:
    n_lines = sum(1 for _ in f)
print(f"Found QA pairs: {n_lines} in {QA_JSONL.name}")

# ---- Tokenizer/model init (defer model weights to training cell if you prefer) ----
tokenizer = AutoTokenizer.from_pretrained(CFG["base_model"])
# Ensure PAD token exists for some T5 checkpoints
if tokenizer.pad_token is None:
    tokenizer.pad_token = tokenizer.eos_token
tokenizer.padding_side = "right"  # more stable with fp16/bf16

# We'll instantiate the model in the next cell right before training to free VRAM for preprocessing.
```

```
Using device: cpu | bf16=False fp16=False
Found QA pairs: 50 in qa_pairs.jsonl
```

```
!pip install evaluate
```

```
Collecting evaluate
  Downloading evaluate-0.4.5-py3-none-any.whl.metadata (9.5 kB)
Requirement already satisfied: datasets>=2.0.0 in /usr/local/lib/python3.12/dist-packages (from evaluate) (4.0.0)
Requirement already satisfied: numpy>=1.17 in /usr/local/lib/python3.12/dist-packages (from evaluate) (2.0.2)
Requirement already satisfied: dill in /usr/local/lib/python3.12/dist-packages (from evaluate) (0.3.8)
Requirement already satisfied: pandas in /usr/local/lib/python3.12/dist-packages (from evaluate) (2.2.2)
Requirement already satisfied: requests>=2.19.0 in /usr/local/lib/python3.12/dist-packages (from evaluate) (2.32.4)
Requirement already satisfied: tqdm>=4.62.1 in /usr/local/lib/python3.12/dist-packages (from evaluate) (4.67.1)
Requirement already satisfied: xxhash in /usr/local/lib/python3.12/dist-packages (from evaluate) (3.5.0)
Requirement already satisfied: multiprocess in /usr/local/lib/python3.12/dist-packages (from evaluate) (0.70.16)
Requirement already satisfied: fsspec>=2021.05.0 in /usr/local/lib/python3.12/dist-packages (from fsspec[http]>=2021
Requirement already satisfied: huggingface-hub>=0.7.0 in /usr/local/lib/python3.12/dist-packages (from evaluate) (0.
Requirement already satisfied: packaging in /usr/local/lib/python3.12/dist-packages (from evaluate) (25.0)
```

```
Requirement already satisfied: filelock in /usr/local/lib/python3.12/dist-packages (from datasets>=2.0.0->evaluate)
Requirement already satisfied: pyarrow>=15.0.0 in /usr/local/lib/python3.12/dist-packages (from datasets>=2.0.0->eva
Requirement already satisfied: pyyaml>=5.1 in /usr/local/lib/python3.12/dist-packages (from datasets>=2.0.0->evaluat
Requirement already satisfied: aiohttp!=4.0.0a0,!=4.0.0a1 in /usr/local/lib/python3.12/dist-packages (from fsspec[ht
Requirement already satisfied: typing-extensions>=3.7.4.3 in /usr/local/lib/python3.12/dist-packages (from huggingfa
Requirement already satisfied: hf-xet<2.0.0,>=1.1.3 in /usr/local/lib/python3.12/dist-packages (from huggingface-hub
Requirement already satisfied: charset_normalizer<4,>=2 in /usr/local/lib/python3.12/dist-packages (from requests>=2
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.12/dist-packages (from requests>=2.19.0->evalu
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.12/dist-packages (from requests>=2.19.0-
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.12/dist-packages (from requests>=2.19.0-
Requirement already satisfied: python-dateutil>=2.8.2 in /usr/local/lib/python3.12/dist-packages (from pandas->evalu
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.12/dist-packages (from pandas->evaluate) (2025
Requirement already satisfied: tzdata>=2022.7 in /usr/local/lib/python3.12/dist-packages (from pandas->evaluate) (20
Requirement already satisfied: aiohappyeyeballs>=2.5.0 in /usr/local/lib/python3.12/dist-packages (from aiohttp!=4.0
Requirement already satisfied: aiosignal>=1.4.0 in /usr/local/lib/python3.12/dist-packages (from aiohttp!=4.0.0a0,!=
Requirement already satisfied: attrs>=17.3.0 in /usr/local/lib/python3.12/dist-packages (from aiohttp!=4.0.0a0,!=4.0
Requirement already satisfied: frozenlist>=1.1.1 in /usr/local/lib/python3.12/dist-packages (from aiohttp!=4.0.0a0,
Requirement already satisfied: multidict<7.0,>=4.5 in /usr/local/lib/python3.12/dist-packages (from aiohttp!=4.0.0a0
Requirement already satisfied: propcache>=0.2.0 in /usr/local/lib/python3.12/dist-packages (from aiohttp!=4.0.0a0,!=
Requirement already satisfied: yarl<2.0,>=1.17.0 in /usr/local/lib/python3.12/dist-packages (from aiohttp!=4.0.0a0,
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.12/dist-packages (from python-dateutil>=2.8.2->pan
Downloading evaluate-0.4.5-py3-none-any.whl (84 kB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 84.1/84.1 kB 2.3 MB/s eta 0:00:00
Installing collected packages: evaluate
Successfully installed evaluate-0.4.5
```

Load Q/A Dataset

```
def load_qa_jsonl(path: Path) -> List[Dict[str, Any]]:
    rows = []
    with open(path, "r", encoding="utf-8") as f:
        for line in f:
            rows.append(json.loads(line))
    return rows

# Expect columns: question, answer, metric, year, company, ...
qa_rows = load_qa_jsonl(QA_JSONL)
```

```
df = pd.DataFrame(qa_rows)
print("Total Q/A loaded:", len(df))
df.head(5)
```

```
Total Q/A loaded: 50
```

	question	answer	metric	year	company	source_doc	context_snippet	confidence_heuristic
0	What was Infosys's revenue?	89,032	revenue	None	Infosys	annual-report-2024	(%) Revenue from operations 1,28,933 1,24,014 ...	0.77
1	What was Infosys's net income in 2025?	26,713	net_income	2025	Infosys	infosys-ar-25	w hedges Other items of other comprehensive in...	0.92
2	What was Infosys's operating cash flow?	29,022	cash_flow	None	Infosys	annual-report-2024	and unbilled revenue (2,933) (5,065) Loans, ot...	0.77

Train/Val/Test Split & Formatting

```
# We'll do 80/10/10 split
df = df.sample(frac=1.0, random_state=SEED).reset_index(drop=True)
n = len(df)
n_train = int(0.8*n); n_val = int(0.1*n)
train_df = df.iloc[:n_train].copy()
val_df   = df.iloc[n_train:n_train+n_val].copy()
test_df  = df.iloc[n_train+n_val:].copy()

def to_sft_format(df: pd.DataFrame) -> Dataset:
    # Prompt style: "question: <Q>"
    prompts = [f"question: {q}" for q in df["question"].tolist()]
    targets = [str(a) for a in df["answer"].tolist()]
```

```
    return Dataset.from_dict({"prompt": prompts, "target": targets})

ds_train = to_sft_format(train_df)
ds_val   = to_sft_format(val_df)
ds_test  = to_sft_format(test_df)

raw_ds = DatasetDict(train=ds_train, validation=ds_val, test=ds_test)
raw_ds
```

```
DatasetDict({
    train: Dataset({
        features: ['prompt', 'target'],
        num_rows: 40
    })
    validation: Dataset({
        features: ['prompt', 'target'],
        num_rows: 5
    })
    test: Dataset({
        features: ['prompt', 'target'],
        num_rows: 5
    })
})
```

Tokenization

```
# Use the tokenizer already created from CFG["base_model"]
# tokenizer = AutoTokenizer.from_pretrained(CFG["base_model"])  # <-- already done earlier

MAX_INPUT_TOKENS  = CFG["max_input_len"]
MAX_TARGET_TOKENS = CFG["max_target_len"]

def preprocess(batch):
    # Encode inputs
    enc = tokenizer(
        batch["prompt"],
```

```
            max_length=MAX_INPUT_TOKENS,
            truncation=True,
            padding=False,              # collator will pad dynamically
        )
        # Encode targets (new API; avoids deprecated as_target_tokenizer)
        with_targets = tokenizer(
            text_target=batch["target"],
            max_length=MAX_TARGET_TOKENS,
            truncation=True,
            padding=False,
        )
        enc["labels"] = with_targets["input_ids"]
        return enc

    tokenized = raw_ds.map(
        preprocess,
        batched=True,
        remove_columns=raw_ds["train"].column_names,
        desc="Tokenizing",
    )

    # Dynamic padding at batch time; uses pad_token_id from tokenizer/model
    data_collator = DataCollatorForSeq2Seq(
        tokenizer=tokenizer,
        model=CFG["base_model"],
        padding="longest",
    )
    tokenized
```

```
Tokenizing: 100%  40/40 [00:00<00:00, 305.79 examples/s]
Tokenizing: 100%  5/5 [00:00<00:00, 62.77 examples/s]
Tokenizing: 100%  5/5 [00:00<00:00, 75.09 examples/s]
DatasetDict({
    train: Dataset({
        features: ['input_ids', 'attention_mask', 'labels'],
        num_rows: 40
    })
    validation: Dataset({
        features: ['input_ids', 'attention_mask', 'labels'],
        num_rows: 5
    })
    test: Dataset({
        features: ['input_ids', 'attention_mask', 'labels'],
        num_rows: 5
    })
})
```

Baseline (Pre-Fine-Tuning) Evaluation

```
# ---- Baseline (pre-finetuning) inference & eval — improved ----
# Uses the same CFG, tokenizer, device as earlier cells.

# Load base model (eval mode; safe pads)
base_model = AutoModelForSeq2SeqLM.from_pretrained(CFG["base_model"])
base_model.to(device)
base_model.eval()
if tokenizer.pad_token_id is None and tokenizer.eos_token_id is not None:
    tokenizer.pad_token = tokenizer.eos_token

def _is_numeric_intent(q: str) -> bool:
    ql = (q or "").lower()
    return any(k in ql for k in ("revenue", "net profit", "net income", "eps", "ebitda", "assets", "liabilities", '
```

```
def _build_prompt(q: str) -> str:
    # Keep your original “question:” style but bias concise numeric answers when relevant
    if _is_numeric_intent(q):
        return (
            "You are a financial QA assistant. "
            "Return ONLY the numeric figure with currency/unit if present; if unknown say 'Not in scope'. "
            f"question: {q}"
        )
    return f"question: {q}"

@torch.inference_mode()
def generate_answers(model, questions: List[str], max_new_tokens: int = 64, temp: float = 0.0, batch_size: int = 1(
    """
    Vectorized generation with per-item latency. Returns list[(answer, latency_s)].
    """
    do_sample = temp > 0.0
    gen_kwargs = dict(
        max_new_tokens=int(max_new_tokens),
        do_sample=bool(do_sample),
        temperature=float(temp) if do_sample else None,
        num_beams=1 if do_sample else 1,  # keep deterministic unless you want beams
        pad_token_id=tokenizer.pad_token_id,
        eos_token_id=tokenizer.eos_token_id,
    )
    # remove None to avoid HF warnings
    gen_kwargs = {k: v for k, v in gen_kwargs.items() if v is not None}

    outs: List[Tuple[str, float]] = []
    # batch over questions
    for i in range(0, len(questions), batch_size):
        batch_qs = questions[i:i+batch_size]
        prompts = [_build_prompt(q) for q in batch_qs]

        t0 = time.time()
        enc = tokenizer(
            prompts,
            return_tensors="pt",
            padding=True,
```

```
                truncation=True,
                max_length=min(CFG["max_input_len"], 512),  # safety cap for T5 family
            ).to(device)

            gen_ids = model.generate(**enc, **gen_kwargs)
            t1 = time.time()
            # decode individually; split shared batch latency evenly as an approximation
            batch_latency = (t1 - t0) / max(1, len(batch_qs))
            decoded = tokenizer.batch_decode(gen_ids, skip_special_tokens=True)

            # post-process numeric-intent prompts: strip to first amount if model rambles
            amount_re = re.compile(r"(?:₹|rs\.?|inr|usd|\$|eur|€|gbp|£)?\s?-?(?:\d{1,3}(?:,\d{3})+|\d+(?:\.\d+)?)(?:\s?
            for q, ans in zip(batch_qs, decoded):
                a = ans.strip()
                if _is_numeric_intent(q):
                    m = amount_re.search(a)
                    if m:
                        a = m.group(0).strip()
                    elif "not in scope" in a.lower():
                        a = "Not in scope"
                outs.append((a, batch_latency))
        return outs

    # Take up to 10 test questions for baseline
    sample_test = test_df.head(min(10, len(test_df)))
    qs   = sample_test["question"].tolist()
    gts  = sample_test["answer"].tolist()

    base_preds = generate_answers(base_model, qs, max_new_tokens=CFG["max_target_len"], temp=0.0, batch_size=16)

    # --- Simple normalization for EM/F1 (unchanged logic) ---
    def normalize_text(s):
        s = s.lower().strip()
        s = re.sub(r"[\s]+", " ", s)
        s = re.sub(r"[^a-z0-9\.\-\$₹% ]", "", s)
        return s

    def exact_match(pred, gold):
```

```
    return int(normalize_text(pred) == normalize_text(gold))

def f1_score(pred, gold):
    p = normalize_text(pred).split()
    g = normalize_text(gold).split()
    if not p and not g: return 1.0
    if not p or not g: return 0.0
    common = {}
    for tok in p:
        common[tok] = min(p.count(tok), g.count(tok))
    num_same = sum(common.values())
    if num_same == 0: return 0.0
    precision = num_same / len(p); recall = num_same / len(g)
    return 2*precision*recall/(precision+recall)

em_list, f1_list, times = [], [], []
for (pred, dt), gold in zip(base_preds, gts):
    em_list.append(exact_match(pred, gold))
    f1_list.append(f1_score(pred, gold))
    times.append(dt)

print("Baseline | EM:", np.mean(em_list), "F1:", np.mean(f1_list), "Avg latency (s):", np.mean(times))
pd.DataFrame({
    "question": qs,
    "gold": gts,
    "pred": [p for p,_ in base_preds],
    "latency_s": times,
    "EM": em_list,
    "F1": f1_list
})
```

```
Baseline | EM: 0.0 F1: 0.0 Avg latency (s): 4.030944967269898
```

	question	gold	pred	latency_s	EM	F1
0	What was Infosys's total equity in 2024?	(1)		4.030945	0	0.0
1	What was Infosys's total employee headcount in...	14	900	4.030945	0	0.0
2	Who is the statutory auditor of Infosys?	Makarand M. Joshi & Co., Company Secretaries, ...	samuel sahib	4.030945	0	0.0
3	Who was the CFO of Infosys?	A.G.S. Manikantha Company Secretary	alan sahib	4.030945	0	0.0
4	What was Infosys's net profit in 2024?	25,568	Not in scope	4.030945	0	0.0

Fine-Tuning with HuggingFace Trainer

```
# ----------------------------
# Baseline (unchanged inputs/outputs)
# ----------------------------
# Load base model (no fine-tuning yet)
base_model = AutoModelForSeq2SeqLM.from_pretrained(CFG["base_model"]).to(device)

def generate_answers(model, questions: List[str], max_new_tokens=64, temp=0.0):
    outs = []
    model.eval()
    for q in questions:
        prompt = f"question: {q}"
        enc = tokenizer(prompt, return_tensors="pt").to(device)
        t0 = time.time()
        out = model.generate(
            **enc,
            max_new_tokens=max_new_tokens,
            temperature=temp,
            do_sample=(temp > 0.0),
            pad_token_id=getattr(tokenizer, "pad_token_id", None),
            eos_token_id=getattr(tokenizer, "eos_token_id", None),
        )
```

```
            dt = time.time() - t0
            ans = tokenizer.decode(out[0], skip_special_tokens=True).strip()
            outs.append((ans, dt))
        return outs

    # Take up to 10 test questions for baseline
    sample_test = test_df.head(min(10, len(test_df)))
    base_preds = generate_answers(base_model, sample_test["question"].tolist())

    # Simple normalization for EM/F1
    def normalize_text(s):
        s = s.lower().strip()
        s = re.sub(r"[\s]+", " ", s)
        s = re.sub(r"[^a-z0-9\.\-\$₹% ]", "", s)
        return s

    def exact_match(pred, gold):
        return int(normalize_text(pred) == normalize_text(gold))

    def f1_score(pred, gold):
        # token-level F1 on normalized strings
        p = normalize_text(pred).split()
        g = normalize_text(gold).split()
        if not p and not g:
            return 1.0
        if not p or not g:
            return 0.0
        common = {}
        for tok in p:
            common[tok] = min(p.count(tok), g.count(tok))
        num_same = sum(common.values())
        if num_same == 0:
            return 0.0
        precision = num_same / len(p)
        recall = num_same / len(g)
        return 2 * precision * recall / (precision + recall)

    em_list, f1_list, times = [], [], []
```

```
for (pred, dt), gold in zip(base_preds, sample_test["answer"].tolist()):
    em_list.append(exact_match(pred, gold))
    f1_list.append(f1_score(pred, gold))
    times.append(dt)

print("Baseline | EM:", np.mean(em_list), "F1:", np.mean(f1_list), "Avg latency (s):", np.mean(times))
pd.DataFrame({
    "question": sample_test["question"].tolist(),
    "gold": sample_test["answer"].tolist(),
    "pred": [p for p,_ in base_preds],
    "latency_s": times,
    "EM": em_list,
    "F1": f1_list
})

# ----------------------------
# Added: Fine-tuning (does NOT change the inputs/outputs above)
# ----------------------------
from transformers import TrainingArguments, Trainer, DataCollatorForSeq2Seq, EarlyStoppingCallback

# Safety checks (assumes you prepared 'tokenized' in Step 3 preprocessing)
assert "tokenized" in globals(), "tokenized DatasetDict not found. Build it in Step 3 preprocessing."
assert "train" in tokenized and len(tokenized["train"]) > 0, "tokenized['train'] is missing or empty."

# 1) Trainable model
ft_model = AutoModelForSeq2SeqLM.from_pretrained(CFG["base_model"]).to(device)

# 2) Collator
data_collator = DataCollatorForSeq2Seq(tokenizer=tokenizer, model=ft_model, padding="longest")

# 3) Metrics for Trainer
def compute_metrics(eval_pred):
    import numpy as np, re
    preds, labels = eval_pred

    # preds are logits → convert to ids
    if isinstance(preds, tuple):
        preds = preds[0]
```

```
        pred_ids = np.argmax(preds, axis=-1)

        decoded_preds = tokenizer.batch_decode(pred_ids, skip_special_tokens=True)

        labels = np.where(labels != -100, labels, tokenizer.pad_token_id)
        decoded_labels = tokenizer.batch_decode(labels, skip_special_tokens=True)

        def norm(s):
            s = s.lower().strip()
            s = re.sub(r"\s+", " ", s)
            s = re.sub(r"[^a-z0-9\.\-\$₹% ]", "", s)
            return s

        def em(a, b): return float(norm(a) == norm(b))

        def f1(a, b):
            pa, pb = norm(a).split(), norm(b).split()
            if not pa and not pb: return 1.0
            if not pa or not pb:  return 0.0
            inter = {}
            for t in pa: inter[t] = min(pa.count(t), pb.count(t))
            n = sum(inter.values())
            if n == 0: return 0.0
            p, r = n / len(pa), n / len(pb)
            return 2 * p * r / (p + r)

        ems = [em(p, g) for p, g in zip(decoded_preds, decoded_labels)]
        f1s = [f1(p, g) for p, g in zip(decoded_preds, decoded_labels)]
        return {"em": float(np.mean(ems)), "f1": float(np.mean(f1s))}


    # 4) Training args (uses your CFG & mp_kwargs())
    FT_OUT = Path.cwd() / "fine_tuned_model"
    FT_OUT.mkdir(parents=True, exist_ok=True)

    args = TrainingArguments(
        output_dir=str(FT_OUT),
        overwrite_output_dir=True,
```

```
        num_train_epochs=CFG.get("num_epochs", 5),
        per_device_train_batch_size=CFG.get("train_batch_size", 8),
        per_device_eval_batch_size=CFG.get("eval_batch_size", 16),
        gradient_accumulation_steps=CFG.get("grad_accum_steps", 2),
        learning_rate=CFG.get("lr", 2e-4),
        weight_decay=CFG.get("weight_decay", 0.01),
        warmup_ratio=CFG.get("warmup_ratio", 0.06),
        lr_scheduler_type=CFG.get("lr_scheduler", "cosine"),
        eval_strategy=CFG.get("eval_strategy", "steps"),
        eval_steps=CFG.get("eval_steps", 200),
        save_steps=CFG.get("save_steps", 200),
        save_total_limit=CFG.get("save_total_limit", 2),
        logging_steps=CFG.get("logging_steps", 50),
        # predict_with_generate=True,
        fp16=mp_kwargs().get("fp16", False),
        bf16=mp_kwargs().get("bf16", False),
        report_to=CFG.get("report_to", "none"),
        load_best_model_at_end=True,
        metric_for_best_model="f1",
        greater_is_better=True,
        gradient_checkpointing=CFG.get("gradient_checkpointing", True),
    )

    callbacks = [EarlyStoppingCallback(early_stopping_patience=CFG.get("early_stop_patience", 3))]

    trainer = Trainer(
        model=ft_model,
        args=args,
        train_dataset=tokenized["train"],
        eval_dataset=tokenized.get("validation", tokenized["train"]),
        data_collator=data_collator,
        tokenizer=tokenizer,
        compute_metrics=compute_metrics,
        callbacks=callbacks,
    )

    train_result = trainer.train()
    _ = trainer.evaluate()
```

```
# Save FT model
trainer.save_model(FT_OUT)
tokenizer.save_pretrained(FT_OUT)

# ---------------------------
# Evaluate the fine-tuned model (extra print; does not change your original outputs)
# ---------------------------
ft_model = AutoModelForSeq2SeqLM.from_pretrained(FT_OUT).to(device).eval()
ft_preds = generate_answers(ft_model, sample_test["question"].tolist(), max_new_tokens=64, temp=0.0)

ft_em, ft_f1, ft_times = [], [], []
for (pred, dt), gold in zip(ft_preds, sample_test["answer"].tolist()):
    ft_em.append(exact_match(pred, gold))
    ft_f1.append(f1_score(pred, gold))
    ft_times.append(dt)

print("Fine-tuned | EM:", np.mean(ft_em), "F1:", np.mean(ft_f1), "Avg latency (s):", np.mean(ft_times))
# (Optional) ft_df = pd.DataFrame({...})  # intentionally not returned to preserve original outputs
```

```
The following generation flags are not valid and may be ignored: ['temperature']. Set `TRANSFORMERS_VERBOSITY=info`
The following generation flags are not valid and may be ignored: ['temperature']. Set `TRANSFORMERS_VERBOSITY=info`
The following generation flags are not valid and may be ignored: ['temperature']. Set `TRANSFORMERS_VERBOSITY=info`
The following generation flags are not valid and may be ignored: ['temperature']. Set `TRANSFORMERS_VERBOSITY=info`
The following generation flags are not valid and may be ignored: ['temperature']. Set `TRANSFORMERS_VERBOSITY=info`
Baseline | EM: 0.0 F1: 0.0 Avg latency (s): 3.912558364868164
/tmp/ipython-input-1126552302.py:161: FutureWarning: `tokenizer` is deprecated and will be removed in version 5.0.0
  trainer = Trainer(
/usr/local/lib/python3.12/dist-packages/torch/utils/data/dataloader.py:666: UserWarning: 'pin_memory' argument is se
  warnings.warn(warn_msg)
```

```
[15/15 03:27, Epoch 5/5]
```

Step	Training Loss	Validation Loss

```
/usr/local/lib/python3.12/dist-packages/torch/utils/data/dataloader.py:666: UserWarning: 'pin_memory' argument is se
  warnings.warn(warn_msg)
```

```
[1/1 : < :]
The following generation flags are not valid and may be ignored: ['temperature']. Set `TRANSFORMERS_VERBOSITY=info`
The following generation flags are not valid and may be ignored: ['temperature']. Set `TRANSFORMERS_VERBOSITY=info`
The following generation flags are not valid and may be ignored: ['temperature']. Set `TRANSFORMERS_VERBOSITY=info`
The following generation flags are not valid and may be ignored: ['temperature']. Set `TRANSFORMERS_VERBOSITY=info`
The following generation flags are not valid and may be ignored: ['temperature']. Set `TRANSFORMERS_VERBOSITY=info`
Fine-tuned | EM: 0.0 F1: 0.0 Avg latency (s): 0.621498966217041
```

Evaluate Fine-Tuned Model

```
ft_model = AutoModelForSeq2SeqLM.from_pretrained(str(FT_OUT)).to(device)

ft_preds = generate_answers(ft_model, sample_test["question"].tolist(), max_new_tokens=64, temp=0.0)

em_list, f1_list, times = [], [], []
for (pred, dt), gold in zip(ft_preds, sample_test["answer"].tolist()):
    em_list.append(exact_match(pred, gold))
    f1_list.append(f1_score(pred, gold))
    times.append(dt)
```

```
print("Fine-Tuned | EM:", np.mean(em_list), "F1:", np.mean(f1_list), "Avg latency (s):", np.mean(times))
pd.DataFrame({
    "question": sample_test["question"].tolist(),
    "gold": sample_test["answer"].tolist(),
    "pred": [p for p,_ in ft_preds],
    "latency_s": times,
    "EM": em_list,
    "F1": f1_list
})
```

```
The following generation flags are not valid and may be ignored: ['temperature']. Set `TRANSFORMERS_VERBOSITY=info`
The following generation flags are not valid and may be ignored: ['temperature']. Set `TRANSFORMERS_VERBOSITY=info`
The following generation flags are not valid and may be ignored: ['temperature']. Set `TRANSFORMERS_VERBOSITY=info`
The following generation flags are not valid and may be ignored: ['temperature']. Set `TRANSFORMERS_VERBOSITY=info`
The following generation flags are not valid and may be ignored: ['temperature']. Set `TRANSFORMERS_VERBOSITY=info`
Fine-Tuned | EM: 0.0 F1: 0.0 Avg latency (s): 0.3777732849121094
```

	question	gold	pred	latency_s	EM	F1
0	What was Infosys's total equity in 2024?	(1)	68	0.372076	0	0.0
1	What was Infosys's total employee headcount in...	14	2,069	0.395944	0	0.0
2	Who is the statutory auditor of Infosys?	Makarand M. Joshi & Co., Company Secretaries, ...	Registrar	0.369973	0	0.0
3	Who was the CEO of Infosys?	A.G.S. Manikantha Company Secretary	Managing	0.373002	0	0.0

Advanced Fine-Tuning Technique ⟶ Mixture-of-Experts (MoE) - Two-expert LoRA approach with a lightweight router

Two LoRA experts (numeric-focused and textual-focused) trained with PEFT on top of FLAN-T5.

A tiny router (scikit-learn LogisticRegression) that chooses which expert to use per question.

(Optional) Soft routing: generate with both experts and pick the answer with higher router probability or higher token-logprob.

```
# If needed:
# !pip install transformers accelerate sentencepiece peft datasets scikit-learn

import os, re, time, json, random
import numpy as np
import pandas as pd
from pathlib import Path
from typing import List, Dict, Any

import torch
from datasets import Dataset
from transformers import (AutoTokenizer, AutoModelForSeq2SeqLM,
                          DataCollatorForSeq2Seq, TrainingArguments, Trainer)
from peft import LoraConfig, get_peft_model, PeftModel, PeftConfig
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import LogisticRegression
from joblib import dump, load

device = "cuda" if torch.cuda.is_available() else "cpu"
BASE_MODEL = "google/flan-t5-base"
SEED = 42
random.seed(SEED); np.random.seed(SEED); torch.manual_seed(SEED)

FT_DIR = ROOT / "fine_tuned_model"
FT_DIR.mkdir(parents=True, exist_ok=True)

print("Device:", device)
```

```
Device: cpu
```

Prepare Splits (Numeric vs Textual)

```
import re
import pandas as pd
from datasets import Dataset

# Assumes df exists from Step 3 (columns: question, answer, ...)
assert "df" in globals(), "Run your Step 3 loading to create `df` (Q/A dataframe)."

# Reproducibility
if "SEED" not in globals():
    SEED = 42

# 1) Clean & sanity checks
_df = (
    df.copy()
      .rename(columns={"question": "question", "answer": "answer"})  # explicit
      .dropna(subset=["question", "answer"])
)

# Trim whitespace
_df["question"] = _df["question"].astype(str).str.strip()
_df["answer"]   = _df["answer"].astype(str).str.strip()

# De-duplicate by (question, answer) to avoid leakage/memorization
_df = _df.drop_duplicates(subset=["question", "answer"]).reset_index(drop=True)

# 2) Improved expert labeling (numeric vs textual) using BOTH question & answer
def is_numeric_like(question: str, answer: str) -> bool:
    q = str(question).lower()
    a = str(answer).lower()

    # Numeric cues in answer
    ans_has_digit = bool(re.search(r"\d", a))
    ans_has_units = any(u in a for u in ["%", "percent", "₹", "$", "rs.", "rupees", "kg", "km", "m", "cm", "°", "ye

    # Numeric cues in question
    q_has_digit   = bool(re.search(r"\d", q))
    q_has_ops     = bool(re.search(r"[+\-*/=]", q))
    q_has_words   = any(w in q for w in [
```

```
            "sum","difference","average","mean","median","mode","rate","ratio","percent","increase","decrease",
            "profit","loss","interest","distance","speed","time","area","perimeter","approx","round","total","count","l
        ])
        q_has_symbols = any(s in q for s in ["₹", "$", "%"])

        score = (
            2*ans_has_digit + ans_has_units +
            2*q_has_words + q_has_ops + q_has_symbols + q_has_digit
        )
        # Threshold (tuned to be slightly conservative):
        return score >= 2

    # 3) Split into numeric/textual groups
    is_num_mask = _df.apply(lambda r: is_numeric_like(r["question"], r["answer"]), axis=1)
    numeric_df  = _df[is_num_mask].reset_index(drop=True)
    textual_df  = _df[~is_num_mask].reset_index(drop=True)

    print("Numeric QAs:", len(numeric_df), "Textual QAs:", len(textual_df))

    # 4) Safe 80/20 split per group (handles tiny datasets)
    def safe_split(d: pd.DataFrame, frac=0.8, seed=SEED):
        n = len(d)
        if n == 0:
            return d.copy(), d.copy()  # both empty
        if n == 1:
            return d.copy(), d.iloc[0:0].copy()  # all train, empty val
        # Make sure we always get at least 1 in val when n>=2
        train = d.sample(frac=frac, random_state=seed)
        val   = d.drop(train.index)
        if len(val) == 0:
            # force-move one row to val for stability
            val = train.sample(n=1, random_state=seed)
            train = train.drop(val.index)
        return train.reset_index(drop=True), val.reset_index(drop=True)

    num_train, num_val = safe_split(numeric_df, frac=0.8, seed=SEED)
    txt_train, txt_val = safe_split(textual_df, frac=0.8, seed=SEED)
```

```
# 5) Convert to simple SFT-style datasets (unchanged output schema)
def to_sft(ds_df: pd.DataFrame) -> Dataset:
    return Dataset.from_dict({
        "prompt": [f"question: {q}" for q in ds_df["question"].tolist()],
        "target": [str(a) for a in ds_df["answer"].tolist()]
    })

ds_num_train = to_sft(num_train)
ds_num_val   = to_sft(num_val)
ds_txt_train = to_sft(txt_train)
ds_txt_val   = to_sft(txt_val)
```

```
Numeric QAs: 30 Textual QAs: 20
```

Tokenization & Collator

```
# --- Improved tokenization (same outputs) ---

import re
import numpy as np
from transformers import AutoTokenizer, DataCollatorForSeq2Seq

tokenizer = AutoTokenizer.from_pretrained(BASE_MODEL)

MAX_INPUT_TOKENS  = 256
MAX_TARGET_TOKENS = 64

_space_rx = re.compile(r"\s+")

def _clean_text_list(xs):
    # robust string clean for batched lists
    out = []
    for x in xs:
        s = "" if x is None else str(x)
        s = _space_rx.sub(" ", s.strip())
```

```
            out.append(s)
        return out

    def preprocess(batch):
        # 1) Clean inputs/targets
        prompts = _clean_text_list(batch["prompt"])
        targets = _clean_text_list(batch["target"])

        # 2) Tokenize inputs (pad/truncate to fixed length for stable training)
        model_inputs = tokenizer(
            prompts,
            max_length=MAX_INPUT_TOKENS,
            truncation=True,
            padding="max_length",
            return_attention_mask=True,
        )

        # 3) Tokenize labels (pad/truncate to fixed length)
        #    Use as_target_tokenizer for older transformers compatibility
        with tokenizer.as_target_tokenizer():
            labels = tokenizer(
                targets,
                max_length=MAX_TARGET_TOKENS,
                truncation=True,
                padding="max_length",
            )

        # 4) Replace pad_token_id in labels with -100 so loss ignores padding
        pad_id = tokenizer.pad_token_id
        processed_labels = []
        for seq in labels["input_ids"]:
            processed_labels.append([(tok if tok != pad_id else -100) for tok in seq])

        model_inputs["labels"] = processed_labels
        return model_inputs

    num_train_tk = ds_num_train.map(
        preprocess, batched=True, remove_columns=ds_num_train.column_names
```

```
)
num_val_tk   = ds_num_val.map(
    preprocess, batched=True, remove_columns=ds_num_val.column_names
)
txt_train_tk = ds_txt_train.map(
    preprocess, batched=True, remove_columns=ds_txt_train.column_names
)
txt_val_tk   = ds_txt_val.map(
    preprocess, batched=True, remove_columns=ds_txt_val.column_names
)

# Collator: longest padding at batch-time; labels already contain -100 where needed
collator = DataCollatorForSeq2Seq(tokenizer=tokenizer, model=BASE_MODEL, padding="longest")
```

```
Map: 100%                                                 24/24 [00:00<00:00, 404.28 examples/s]
/usr/local/lib/python3.12/dist-packages/transformers/tokenization_utils_base.py:4006: UserWarning: `as_target_token
  warnings.warn(
Map: 100%                                                 6/6 [00:00<00:00, 178.92 examples/s]
Map: 100%                                                 16/16 [00:00<00:00, 514.28 examples/s]
Map: 100%                                                 4/4 [00:00<00:00, 136.67 examples/s]
```

LoRA Expert A (Numeric)

```
from peft import LoraConfig, get_peft_model
from transformers import AutoModelForSeq2SeqLM, TrainingArguments, Trainer

# Base backbone for the numeric expert
base_numeric = AutoModelForSeq2SeqLM.from_pretrained(BASE_MODEL).to(device)

# T5/FLAN-T5 friendly target modules
t5_target_modules = ["q", "k", "v", "o", "wi_0", "wi_1", "wo"]
```

```
lora_cfg = LoraConfig(
    r=8,
    lora_alpha=16,
    target_modules=t5_target_modules,
    lora_dropout=0.05,
    bias="none",
    task_type="SEQ_2_SEQ_LM",
)

num_model = get_peft_model(base_numeric, lora_cfg)

if hasattr(num_model, "config"):
    num_model.config.use_cache = False  # must be off when training/checkpointing

# Some older transformers don't auto-enable input grads:
if hasattr(num_model, "enable_input_require_grads"):
    num_model.enable_input_require_grads()

# Keep gradient checkpointing (optional but supported)
if hasattr(num_model, "gradient_checkpointing_enable"):
    num_model.gradient_checkpointing_enable()

# Optional perf knobs (safe no-ops on CPU)
try:
    import torch
    if torch.cuda.is_available():
        torch.backends.cuda.matmul.allow_tf32 = True
except Exception:
    pass

# Mixed precision selection that won't crash on older GPUs
use_bf16 = False
use_fp16 = False
try:
    use_bf16 = torch.cuda.is_available() and torch.cuda.is_bf16_supported()
    use_fp16 = torch.cuda.is_available() and (not use_bf16)
except Exception:
```

```
        pass

args_num = TrainingArguments(
    output_dir=str(FT_DIR / "lora_numeric"),
    learning_rate=5e-5,
    weight_decay=0.01,
    lr_scheduler_type="cosine",
    warmup_ratio=0.06,

    per_device_train_batch_size=8,
    per_device_eval_batch_size=8,
    gradient_accumulation_steps=2,   # steadier updates without bumping VRAM

    num_train_epochs=4,
    # evaluation_strategy="epoch",   # keep commented if your transformers is older
    save_strategy="epoch",
    save_total_limit=2,

    logging_steps=20,
    report_to="none",
    seed=SEED,

    bf16=use_bf16,
    fp16=use_fp16,
    max_grad_norm=1.0,               # prevent exploding grads
    dataloader_pin_memory=torch.cuda.is_available(),
)

# (Nice to have) See what's actually trainable (should be only LoRA params)
try:
    num_model.print_trainable_parameters()
except Exception:
    pass

trainer_num = Trainer(
    model=num_model,
    args=args_num,
    train_dataset=num_train_tk,
```

```
    eval_dataset=num_val_tk,
    data_collator=collator,
    tokenizer=tokenizer,
)

trainer_num.train()

# Save ONLY the LoRA adapter weights + tokenizer for reuse
num_model.save_pretrained(str(FT_DIR / "lora_numeric"))
tokenizer.save_pretrained(str(FT_DIR / "lora_numeric"))
print("Saved LoRA numeric expert at:", (FT_DIR / "lora_numeric").resolve())
```

```
trainable params: 3,391,488 || all params: 250,969,344 || trainable%: 1.3514
/tmp/ipython-input-3081299541.py:81: FutureWarning: `tokenizer` is deprecated and will be removed in version 5.0.0 f
  trainer_num = Trainer(
```

[8/8 08:07, Epoch 4/4]

Step	Training Loss

```
Saved LoRA numeric expert at: /content/drive/MyDrive/RAG-FT-DATA/fine_tuned_model/lora_numeric
```

LoRA Expert B (Textual)

```
from transformers import AutoModelForSeq2SeqLM, TrainingArguments, Trainer

# 1) Load backbone and attach LoRA (reuses your existing lora_cfg)
base_textual = AutoModelForSeq2SeqLM.from_pretrained(BASE_MODEL).to(device)
txt_model = get_peft_model(base_textual, lora_cfg)

# 2) Required for stable T5 training w/ checkpointing (older transformers)
#    - disable KV cache during training
#    - ensure input embeddings require grad so backprop works
if hasattr(txt_model, "config"):
    txt_model.config.use_cache = False
```

```
    if hasattr(txt_model, "enable_input_require_grads"):
        txt_model.enable_input_require_grads()
    if hasattr(txt_model, "gradient_checkpointing_enable"):
        txt_model.gradient_checkpointing_enable()

    # 3) Optional perf knobs (safe no-ops on CPU)
    try:
        if torch.cuda.is_available():
            torch.backends.cuda.matmul.allow_tf32 = True
    except Exception:
        pass

    # 4) Mixed-precision flags that won't crash on older GPUs
    use_bf16, use_fp16 = False, False
    try:
        use_bf16 = torch.cuda.is_available() and torch.cuda.is_bf16_supported()
        use_fp16 = torch.cuda.is_available() and (not use_bf16)
    except Exception:
        pass

    # 5) Training args: warmup, weight decay, cosine schedule, grad-accum, clipping
    args_txt = TrainingArguments(
        output_dir=str(FT_DIR / "lora_textual"),
        learning_rate=5e-5,
        weight_decay=0.01,
        lr_scheduler_type="cosine",
        warmup_ratio=0.06,

        per_device_train_batch_size=8,
        per_device_eval_batch_size=8,
        gradient_accumulation_steps=2,   # steadier updates, lower VRAM

        num_train_epochs=4,
        # evaluation_strategy="epoch",   # keep commented for older transformers
        save_strategy="epoch",
        save_total_limit=2,

        logging_steps=20,
```

```
        report_to="none",
        seed=SEED,

        bf16=use_bf16,
        fp16=use_fp16,
        max_grad_norm=1.0,
        dataloader_pin_memory=torch.cuda.is_available(),
    )

    # (Nice to have) Confirm only LoRA params are trainable
    try:
        txt_model.print_trainable_parameters()
    except Exception:
        pass

    trainer_txt = Trainer(
        model=txt_model,
        args=args_txt,
        train_dataset=txt_train_tk,
        eval_dataset=txt_val_tk,
        data_collator=collator,
        tokenizer=tokenizer,  # OK even with deprecation warning on older versions
    )

    trainer_txt.train()

    # Save ONLY the LoRA adapter + tokenizer for later loading on the same base model
    txt_model.save_pretrained(str(FT_DIR / "lora_textual"))
    tokenizer.save_pretrained(str(FT_DIR / "lora_textual"))
    print("Saved LoRA textual expert at:", (FT_DIR / "lora_textual").resolve())
```

```
trainable params: 3,391,488 || all params: 250,969,344 || trainable%: 1.3514
/tmp/ipython-input-3565904750.py:65: FutureWarning: `tokenizer` is deprecated and will be removed in version 5.0.0 f
  trainer_txt = Trainer(
```

[4/4 04:54, Epoch 4/4]

Step	Training Loss

```
Saved LoRA textual expert at: /content/drive/MyDrive/RAG-FT-DATA/fine_tuned_model/lora_textual
```

```
import re

# Precompile once (keeps inference fast)
_NUM_PATTERNS = [
    # Plain/decimal/scientific with optional sign & separators
    re.compile(r'(?<!\w)[+−]?\d{1,3}(?:[,]\d{2,3})*(?:\.\d+)?(?:e[+−]?\d+)?(?!\w)'),
    # Percentages
    re.compile(r'\d+(?:\.\d+)?\s*%'),
    # Currency (₹, $, €, £) with amount
    re.compile(r'[₹$€£]\s*\d+(?:[,.\d])*'),
    # Fractions
    re.compile(r'\b\d+\s*/\s*\d+\b'),
    # Ranges (10–15, 2 — 3, between 4 and 5)
    re.compile(r'\b\d+\s*[—–]\s*\d+\b'),
    re.compile(r'\bbetween\s+\d+(?:\.\d+)?\s+and\s+\d+(?:\.\d+)?\b'),
    # Ratios
    re.compile(r'\b\d+\s*:\s*\d+\b'),
    # Ordinals (1st, 2nd, 3rd, 4th)
    re.compile(r'\b\d+(st|nd|rd|th)\b'),
    # Times/Dates (simple heuristics: 12:30, 2021, 01/02/2024)
    re.compile(r'\b\d{1,2}:\d{2}\b'),
    re.compile(r'\b\d{4}\b'),
    re.compile(r'\b\d{1,2}[/-]\d{1,2}[/-]\d{2,4}\b'),
    # Units (numbers followed by common units)
    re.compile(r'\d+(?:\.\d+)?\s*(km|m|cm|mm|kg|g|lb|lbs|°c|°f|hrs?|hours?|mins?|minutes?|secs?|seconds?)\b', re.I)
]

# Spelled-out numerals (incl. Indian scale)
```

```
_NUM_WORDS_RX = re.compile(
    r'\b('
    r'zero|one|two|three|four|five|six|seven|eight|nine|ten|'
    r'eleven|twelve|thirteen|fourteen|fifteen|sixteen|seventeen|eighteen|nineteen|'
    r'twenty|thirty|forty|fifty|sixty|seventy|eighty|ninety|'
    r'hundred|thousand|million|billion|lakh|crore|half|quarter'
    r')\b', re.I
)

_BOOL_LIKE = {"yes", "no", "true", "false", "y", "n"}

def is_numeric_answer(s: str) -> bool:
    if s is None:
        return False
    t = str(s).strip()
    if not t:
        return False
    tl = t.lower()

    # Filter common non-numeric short answers
    if tl in _BOOL_LIKE:
        return False

    # Fast path: any digit at all — often numeric
    if any(ch.isdigit() for ch in tl):
        return True

    # Spelled-out numerals (with word boundaries)
    if _NUM_WORDS_RX.search(tl):
        return True

    # Heuristic patterns (currency, %, ranges, ratios, dates/times, units, etc.)
    for rx in _NUM_PATTERNS:
        if rx.search(tl):
            return True

    return False
```

Train a Tiny Router (LogReg)

```
import re
import numpy as np
import pandas as pd
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.model_selection import StratifiedKFold
from sklearn.metrics import f1_score
from sklearn.linear_model import LogisticRegression
from joblib import dump

# Build router df (same columns)
router_df = df[["question", "answer"]].copy()
router_df["label_numeric"] = router_df["answer"].apply(is_numeric_answer).astype(int)

# --- Hygiene: drop NaNs and normalize spaces ---
router_df = router_df.dropna(subset=["question"]).copy()
router_df["question"] = router_df["question"].astype(str).str.strip()
router_df["question"] = router_df["question"].str.replace(r"\s+", " ", regex=True)

# Optional: light normalization (keep currency & operators)
def _preproc(s: str) -> str:
    s = s.lower().strip()
    # normalize common variants
    s = s.replace("rs.", "₹").replace("rupees", "₹")
    # collapse long numbers to a token to reduce sparsity
    s = re.sub(r"\d+(\.\d+)?", " <num> ", s)
    s = re.sub(r"\s+", " ", s)
    return s

# --- Stronger vectorizer but still a single TfidfVectorizer (keeps your downstream code intact) ---
vec = TfidfVectorizer(
    preprocessor=_preproc,              # keep ₹, %, operators via token_pattern below
    ngram_range=(1, 2),
    max_features=10000,                 # a bit larger coverage
```

```
    min_df=2,                          # drop ultra-rare noise
    max_df=0.98,                       # drop near-stopwords
    sublinear_tf=True,
    lowercase=True,
    token_pattern=r"(?u)\b[\w₹$%+\-*/=\.]+\b",
)

X = vec.fit_transform(router_df["question"])
y = router_df["label_numeric"].values

# --- Small CV over C to improve robustness; still returns a plain LogisticRegression ---
skf = StratifiedKFold(n_splits=5, shuffle=True, random_state=SEED if "SEED" in globals() else 42)
best_C, best_score = None, -1.0
for C in [0.25, 0.5, 1.0, 2.0, 4.0]:
    scores = []
    for tr, va in skf.split(X, y):
        Xtr, Xva = X[tr], X[va]
        ytr, yva = y[tr], y[va]
        tmp_clf = LogisticRegression(
            C=C,
            max_iter=2000,
            class_weight="balanced",
            solver="liblinear",   # robust on small/medium sparse problems
            penalty="l2"
        )
        tmp_clf.fit(Xtr, ytr)
        p = tmp_clf.predict(Xva)
        scores.append(f1_score(yva, p))
    mean_f1 = float(np.mean(scores))
    if mean_f1 > best_score:
        best_score, best_C = mean_f1, C

# Train final model with the best C on full data
clf = LogisticRegression(
    C=best_C,
    max_iter=2000,
    class_weight="balanced",
    solver="liblinear",
```

```
    penalty="l2"
)
clf.fit(X, y)

# Persist (same filenames)
dump(vec, FT_DIR / "router_vectorizer.joblib")
dump(clf, FT_DIR / "router_clf.joblib")

# Quick sanity check (same prints)
print("Router class balance:", np.mean(y))
print("Router sample preds:", clf.predict(vec.transform(router_df["question"].head(5))))
```

```
Router class balance: 0.6
Router sample preds: [0 1 0 1 1]
```

Inference with Hard Routing (MoE)

```
# Load base once for inference; attach LoRA adapters once and switch per call
base_for_infer = AutoModelForSeq2SeqLM.from_pretrained(BASE_MODEL).to(device)

NUM_PATH = str(FT_DIR / "lora_numeric")
TXT_PATH = str(FT_DIR / "lora_textual")

from peft import PeftModel

# Wrap base with textual adapter, then load numeric adapter too
_peft_infer = PeftModel.from_pretrained(base_for_infer, TXT_PATH).to(device).eval()
# textual adapter name will be "default" typically
_TEXT_ADAPTER = getattr(_peft_infer, "active_adapter", "default")
# load numeric under a fixed name
_peft_infer.load_adapter(NUM_PATH, adapter_name="numeric")
_NUM_ADAPTER = "numeric"

vectorizer = load(FT_DIR / "router_vectorizer.joblib")
router_clf = load(FT_DIR / "router_clf.joblib")
```

```
def _autocast_dtype():
    if torch.cuda.is_available():
        try:
            return torch.bfloat16 if torch.cuda.is_bf16_supported() else torch.float16
        except Exception:
            return torch.float16
    return None

@torch.no_grad()
def _generate_with_adapter(model, adapter_name: str, question: str, max_new_tokens=64, temperature=0.0):
    """Generate text and (if supported) compute avg token logprob."""
    model.set_adapter(adapter_name)
    model.eval()

    prompt = f"question: {question}"
    enc = tokenizer(prompt, return_tensors="pt").to(device)
    gen_kwargs = {
        "max_new_tokens": max_new_tokens,
        "do_sample": (temperature > 0.0),
        "temperature": (temperature if temperature > 0.0 else None),
        "pad_token_id": getattr(tokenizer, "pad_token_id", None),
        "eos_token_id": getattr(tokenizer, "eos_token_id", None),
    }
    # Remove None entries (older transformers can choke on None)
    gen_kwargs = {k: v for k, v in gen_kwargs.items() if v is not None}

    # try to request scores for log-prob based soft routing (if supported)
    want_scores = True
    try_kwargs = dict(return_dict_in_generate=True, output_scores=True)
    amp_dtype = _autocast_dtype()

    try:
        t0 = time.time()
        if amp_dtype:
            with torch.autocast("cuda", dtype=amp_dtype):
                out = model.generate(**enc, **gen_kwargs, **try_kwargs)
        else:
```

```
                out = model.generate(**enc, **gen_kwargs, **try_kwargs)
            dt = time.time() - t0

            seq = out.sequences[0]
            input_len = enc["input_ids"].shape[1]
            toks = seq[input_len:]
            # compute avg logprob over generated tokens
            lps = []
            for step, tok_id in enumerate(toks):
                step_logits = out.scores[step][0]  # [vocab]
                lp = torch.log_softmax(step_logits, dim=-1)[int(tok_id)].item()
                lps.append(lp)
            avg_lp = float(np.mean(lps)) if lps else -1e9
            text = tokenizer.decode(seq, skip_special_tokens=True).strip()
            return text, dt, avg_lp
        except TypeError:
            # Older transformers: no scores available
            want_scores = False

        t0 = time.time()
        if amp_dtype:
            with torch.autocast("cuda", dtype=amp_dtype):
                out = model.generate(**enc, **gen_kwargs)
        else:
            out = model.generate(**enc, **gen_kwargs)
        dt = time.time() - t0
        text = tokenizer.decode(out[0], skip_special_tokens=True).strip()
        return text, dt, (None if not want_scores else -1e9)

    @torch.no_grad()
    def moe_lora_answer(question: str, max_new_tokens=64, temperature=0.0, soft=False) -> Dict[str, Any]:
        """
        Hard routing (default): choose one expert via router.
        soft=True: generate with both experts and pick by higher avg token log-prob if available;
                   otherwise fall back to router probability threshold (>=0.5 -> numeric).
        """
        # Router predict
        Xq = vectorizer.transform([question])
```

```
        prob_numeric = float(router_clf.predict_proba(Xq)[0, 1])

        start = time.time()

        if not soft:
            choose_numeric = (prob_numeric >= 0.5)
            expert = "numeric" if choose_numeric else "textual"
            adapter = _NUM_ADAPTER if choose_numeric else _TEXT_ADAPTER
            ans, gen_dt, _ = _generate_with_adapter(_peft_infer, adapter, question, max_new_tokens, temperature)
            total_dt = time.time() - start
            return {
                "answer": ans,
                "expert_used": expert,
                "p_numeric": round(prob_numeric, 3),
                "latency_sec": round(total_dt, 3),
            }

        # soft routing: run both and pick best by avg log-prob (if available)
        ans_txt, dt_txt, lp_txt = _generate_with_adapter(_peft_infer, _TEXT_ADAPTER, question, max_new_tokens, temperat
        ans_num, dt_num, lp_num = _generate_with_adapter(_peft_infer, _NUM_ADAPTER,  question, max_new_tokens, temperat

        if lp_txt is not None and lp_num is not None:
            choose_numeric = (lp_num >= lp_txt)
        else:
            choose_numeric = (prob_numeric >= 0.5)

        expert = "numeric" if choose_numeric else "textual"
        ans = ans_num if choose_numeric else ans_txt
        total_dt = time.time() - start

        return {
            "answer": ans,
            "expert_used": expert,
            "p_numeric": round(prob_numeric, 3),
            "latency_sec": round(total_dt, 3),
        }

    # Try it
```

```
examples = [
    "What was the revenue in 2025?",
    "Name the key business segments reported.",
]
for q in examples:
    print(q, "->", moe_lora_answer(q))
```

```
What was the revenue in 2025? -> {'answer': '$ 2.2 billion', 'expert_used': 'numeric', 'p_numeric': 0.543, 'latency_
Name the key business segments reported. -> {'answer': 'telecommunications and energy', 'expert_used': 'textual', 'p
```

Compare FT (single) vs MoE-LoRA

```
# Helper metrics
def normalize_text(s):
    s = s.lower().strip()
    s = re.sub(r"[\s]+", " ", s)
    s = re.sub(r"[^a-z0-9\.\-\$₹% ]", "", s)
    return s

def exact_match(pred, gold): return int(normalize_text(pred) == normalize_text(gold))

def f1_score(pred, gold):
    p = normalize_text(pred).split(); g = normalize_text(gold).split()
    if not p and not g: return 1.0
    if not p or not g: return 0.0
    common = {}
    for tok in p: common[tok] = min(p.count(tok), g.count(tok))
    num_same = sum(common.values())
    if num_same == 0: return 0.0
    precision = num_same / len(p); recall = num_same / len(g)
    return 2*precision*recall/(precision+recall)

# Load the single fine-tuned model (from Step 3) or fall back to base
if "ft_model" not in globals():
```

```
    ft_model = AutoModelForSeq2SeqLM.from_pretrained(str(FT_OUT)).to(device)

@torch.no_grad()
def generate_answers(model, questions: List[str], max_new_tokens=64):
    outs = []
    for q in questions:
        enc = tokenizer(f"question: {q}", return_tensors="pt").to(device)
        t0 = time.time()
        out = model.generate(**enc, max_new_tokens=max_new_tokens)
        dt = time.time() - t0
        ans = tokenizer.decode(out[0], skip_special_tokens=True)
        outs.append((ans.strip(), dt))
    return outs

# Sample test
sample = df.sample(n=min(10, len(df)), random_state=123)
qs = sample["question"].tolist()
golds = sample["answer"].tolist()

# Single FT model (if available)
ft_outs = generate_answers(ft_model, qs)
ft_preds = [a for (a, _) in ft_outs]
ft_times = [t for (_, t) in ft_outs]

# MoE-LoRA
moe_preds, moe_times = [], []
for q in qs:
    out = moe_lora_answer(q)
    moe_preds.append(out["answer"])
    moe_times.append(out["latency_sec"])

def eval_run(preds, times, golds):
    em = np.mean([exact_match(p,g) for p,g in zip(preds, golds)])
    f1 = np.mean([f1_score(p,g) for p,g in zip(preds, golds)])
    lat = float(np.mean(times))
    return em, f1, lat

# ft_em, ft_f1, ft_lat   = eval_run([p for p,_ in ft_preds], [t for _,t in ft_preds], golds)
```

```
# moe_em, moe_f1, moe_lat= eval_run(moe_preds, moe_times, golds)
# Evaluate
ft_em, ft_f1, ft_lat   = eval_run(ft_preds, ft_times, golds)
moe_em, moe_f1, moe_lat= eval_run(moe_preds, moe_times, golds)

pd.DataFrame({
    "metric": ["EM","F1","Latency(s)"],
    "Single FT": [ft_em, ft_f1, ft_lat],
    "MoE-LoRA (router)": [moe_em, moe_f1, moe_lat]
})
```

	metric	Single FT	MoE-LoRA (router)
0	EM	0.000000	0.0000
1	F1	0.066667	0.0000
2	Latency(s)	0.643827	1.1853

Guardrail config & helpers (input + output expectations)

```
import re
from typing import List, Dict, Any

# ========= Finance intent (broader coverage, fewer false OOS) =========
# Covers statements/metrics, governance, corp actions, and FY tokens.
_FINANCE_PATTERNS = [
    # statements & metrics
    r"revenue", r"net\s*sales", r"sales", r"turnover",
    r"net\s*(?:income|profit)", r"(?:operating|op\.?)\s*(?:income|profit)",
    r"\bebitda\b", r"\bebit\b", r"\beps\b",
    r"cash(?:\s*and\s*cash\s*equivalents)?", r"operating\s*cash\s*flow", r"free\s*cash\s*flow",
    r"assets?", r"liabilit(?:y|ies)", r"net\s*worth", r"equity", r"debt",
    r"margin", r"gross\s*margin", r"net\s*margin", r"operating\s*margin",
    r"\bpbt\b", r"\bpat\b", r"dividend", r"capex", r"opex",
```

```
        r"receivables", r"payables", r"working\s*capital",
        r"guidance", r"outlook", r"\bmd&?a\b", r"management\s+discussion",
        r"segment(s)?", r"annual\s*report",
        r"balance\s*sheet", r"income\s*statement", r"statement\s*of\s*operations",
        r"cash\s*flow\s*statement",
        # governance / corp actions
        r"\bceo\b", r"\bcfo\b", r"chief\s+financial\s+officer", r"auditor",
        r"headcount", r"employee[s]?",
        r"\bmerger(s)?\b", r"\bacquisition(s)?\b", r"\bm&?a\b",
        # anchors
        r"\byoy\b", r"\bqoq\b", r"\bfx\b", r"\bforex\b",
    ]
    FINANCE_RE = re.compile(r"\b(?:" + "|".join(_FINANCE_PATTERNS) + r")\b", re.IGNORECASE)

    # FY24 / FY2024 / FY 2023-24
    FY_RE = re.compile(r"\bfy\s*'?(\d{2,4})(?:\s*[-/]\s*(\d{2,4}))?\b", re.IGNORECASE)

    # Truly off-domain queries we should reject early
    _HARD_OOS_RE = re.compile(r"\b(capital of|weather|recipe|lyrics?|translate|distance to)\b", re.IGNORECASE)

    def is_finance_query(query: str) -> bool:
        """
        Return True for finance/corporate queries; False only for clearly irrelevant ones.
        This widens 'finance' to include CFO/mergers/FY references so valid queries aren't blocked.
        """
        q = (query or "").lower()
        if _HARD_OOS_RE.search(q):
            return False
        return bool(FINANCE_RE.search(q) or FY_RE.search(q))


    # ========= PII detection (compiled, case-insensitive) =========
    _PII_PATTERNS = [
        r"\bpan\s*number\b", r"\baadhaar\b", r"\bssn\b", r"\bsocial\s+security\b",
        r"\bcredit\s*card\b", r"\bcvv\b", r"\bbank\s*account\b", r"\bifsc\b",
        r"\bphone\s*(number)?\b", r"\bemail\b",
    ]
    PII_RE = re.compile("|".join(_PII_PATTERNS), re.IGNORECASE)
```

```
def contains_pii_request(query: str) -> bool:
    q = (query or "").lower()
    return PII_RE.search(q) is not None


# ========= Numeric expectation (smarter cues, wh-word guard) =========
# Phrases that imply numeric answers; includes "% / growth / how much / value"
NUMERIC_CUE_RE = re.compile(
    r"\b("
    r"revenue|sales|turnover|profit|net profit|eps|ebitda|ebit|margin|guidance|"
    r"growth|%|percent|yoy|qoq|capex|opex|dividend|cash|equity|debt|"
    r"amount|total|figure|number|how much|how many|what(?:'s| is) the (?:value|amount)"
    r")\b",
    re.IGNORECASE,
)

# WH-words that *usually* mean non-numeric, except explicit numeric forms above
_NON_NUMERIC_WH_RE = re.compile(r"^\s*(who|which|where|when|why)\b", re.IGNORECASE)

def expects_numeric_answer(query: str) -> bool:
    """
    Heuristic: True if the question implies a numeric result.
    Exceptions: starts with who/which/where/when/why (unless numeric cues present).
    """
    q = (query or "").lower()
    if NUMERIC_CUE_RE.search(q):
        return True
    # FY/year often implies a numeric metric is being asked (paired with finance terms via is_finance_query)
    if FY_RE.search(q) and is_finance_query(q):
        return True
    # Default: if it starts with non-numeric wh-word, assume non-numeric
    return not bool(_NON_NUMERIC_WH_RE.search(q))


# ========= Numeric-in-answer detection (real-world formats) =========
# Accept currency, %, Indian units, accounting negatives, grouped digits.
_ANS_NUM_RE = re.compile(
```

```
    r"(?:₹|\$|usd|inr|rs\.?)?\s*\(?-?\d{1,3}(?:,\d{3})+|-?\d+(?:\.\d+)?\)?\s*"
    r"(?:%|crore|cr\.?|cr|million|mn|billion|bn|lakh|lakhs|thousand|k)?",
    re.IGNORECASE,
)
YEAR_RE = re.compile(r"\b(19|20)\d{2}\b")

def answer_has_number(ans: str) -> bool:
    """
    True if the answer contains a plausible numeric value (currency/%/units).
    Ignores bare years so '2025' alone doesn't count as a numeric answer.
    """
    s = (ans or "").strip()
    if not s:
        return False
    # any rich numeric span?
    if _ANS_NUM_RE.search(s):
        # make sure it's not *only* a year
        spans = [m.group(0) for m in _ANS_NUM_RE.finditer(s)]
        # keep spans that aren't just a plain year
        for sp in spans:
            # if it has currency/unit/% or commas/decimals/paren negatives → accept
            if re.search(r"(₹|\$|usd|inr|rs\.?|%|crore|cr\.?|cr|million|mn|billion|bn|lakh|lakhs|thousand|k)", sp,
                return True
            if "," in sp or "." in sp or sp.startswith("(") or sp.startswith("-"):
                return True
            if not YEAR_RE.fullmatch(sp):  # plain number but not a 4-digit year
                return True
        return False
    return False
```

Generation with log-prob confidence (for base/finetuned T5)

```
import torch

# expects global: tokenizer, base_model, ft_model (from Step 3)
```

```
def generate_with_scores(model, prompt: str, max_new_tokens=64, temperature=0.0):
    enc = tokenizer(prompt, return_tensors="pt").to(model.device)
    out = model.generate(
        **enc,
        max_new_tokens=max_new_tokens,
        temperature=temperature,
        do_sample=(temperature > 0.0),
        return_dict_in_generate=True,
        output_scores=True
    )
    # Decode answer
    seq_ids = out.sequences[0]
    ans = tokenizer.decode(seq_ids, skip_special_tokens=True).strip()

    # Compute average log-prob for generated tokens (decoder steps only)
    # Align scores to generated tokens (ignore prompt length)
    scores = out.scores  # list[tensor vocab_logits] length = generated_len
    gen_token_ids = seq_ids[len(enc.input_ids[0]):]  # only newly generated token ids
    logprobs = []
    for step_logits, tok_id in zip(scores, gen_token_ids):
        lp = torch.log_softmax(step_logits, dim=-1)[0, tok_id.item()].item()
        logprobs.append(lp)
    avg_logprob = float(sum(logprobs) / max(1, len(logprobs)))

    return ans, avg_logprob, len(gen_token_ids)
```

Optional: reuse RAG's number-consistency check (if available)

```
# Build a map from chunk_id -> full text once (global)
CHUNK_TEXT_BY_ID = {c["chunk_id"]: c["text"] for c in chunks}

def rag_number_consistency_check(answer: str, query: str) -> dict:
    """
    Compare numbers in the answer with numbers found in the *full* retrieved chunks.
    """
```

```
    if "hybrid_retrieve" not in globals():
        return {"used": False, "is_suspicious": False, "suspicious_numbers": [], "contexts": []}

    hits = hybrid_retrieve(query, k_dense=10, k_sparse=10, k_fused=15, k_final=5)
    # Use full texts
    contexts_full = []
    for row in hits.get("reranked_top", []):
        cid = row.get("chunk_id")
        full = CHUNK_TEXT_BY_ID.get(cid, row.get("preview",""))
        contexts_full.append(full)

    # Extract numbers
    ans_nums = set(NUM_RE.findall(answer.replace(",", "")))
    ctx_nums = set()
    for c in contexts_full:
        for n in NUM_RE.findall(c.replace(",", "")):
            ctx_nums.add(n)

    suspicious = sorted(list(ans_nums - ctx_nums))
    return {
        "used": True,
        "is_suspicious": len(suspicious) > 0,
        "suspicious_numbers": suspicious,
        "contexts_checked": len(contexts_full)
    }
```

Guardrailed FT answer wrapper (works for base / finetuned / MoE-LoRA)

```
import time, math, re
from typing import Dict, Any, List

# -------- helpers for numeric extraction from context (preserve currency/units) --------
_NUM_SPAN = re.compile(
    r'(?:₹|\$|usd|inr|rs\.?)?\s*\(?-?\d{1,3}(?:,\d{3})+|-?\d+(?:\.\d+)?\)?\s*'
    r'(?:%|crore|cr\.?|cr|million|mn|billion|bn|lakh|lakhs|thousand|k)?',
    re.IGNORECASE,
```

```
)

def _extract_spans(text: str) -> List[tuple[str,int]]:
    return [(m.group(0).strip(), m.start()) for m in _NUM_SPAN.finditer(text or "")]

def _best_numeric_from_context(query: str, contexts: List[str]) -> str | None:
    """Pick the most relevant numeric span from retrieved contexts."""
    ql = (query or "").lower()
    want_cur = any(k in ql for k in ["revenue","sales","turnover","profit","cash","equity"])
    want_pct = ("margin" in ql) or ("%") in ql
    ym = re.search(r'\b(19|20)\d{2}\b', ql)
    yr = ym.group(0) if ym else None

    best = (None, -1.0)
    for ctx in contexts or []:
        txt = re.sub(r"\s+", " ", str(ctx)).strip()
        spans = _extract_spans(txt)
        if not spans:
            continue
        for span, pos in spans:
            score = 0.0
            # proximity to topical keywords
            for kw in ("revenue","sales","turnover","profit","net profit","margin","cash","equity"):
                m = re.search(kw, txt, re.I)
                if m:
                    score += max(0.0, 1.0 - abs(pos - m.start())/150.0)
            if want_pct and "%" in span: score += 0.6
            if want_cur and re.search(r'(₹|\$|usd|inr|rs\.?)', span, re.I): score += 0.6
            if yr and yr in txt: score += 0.4
            # bonus for specificity (more digits)
            score += min(0.5, len(re.sub(r"[^\d]", "", span))/10.0)
            if score > best[1]:
                best = (span, score)
    return best[0]


def ft_guardrailed_answer(query: str, mode: str = "finetuned",
                          max_new_tokens: int = 64, temperature: float = 0.0,
```

```
                        use_rag_consistency: bool = True) -> Dict[str, Any]:
    t0 = time.time()
    flags: List[str] = []

    # 1) INPUT GUARDRAIL
    if contains_pii_request(query):
        return {
            "method": f"FT:{mode}",
            "answer": "Out of scope (PII request is not allowed).",
            "confidence": 0.2,
            "latency_sec": round(time.time()-t0, 3),
            "flags": ["input_pii_block"]
        }
    if not is_finance_query(query):
        return {
            "method": f"FT:{mode}",
            "answer": "Out of scope (non-financial query).",
            "confidence": 0.3,
            "latency_sec": round(time.time()-t0, 3),
            "flags": ["input_out_of_scope"]
        }

    # 2) INFERENCE
    expectation_numeric = expects_numeric_answer(query)
    prompt = f"question: {query}".strip()

    model_ans = None
    conf_model = 0.0

    if mode == "base":
        assert "base_model" in globals(), "Load base_model first (Step 3)."
        model_ans, conf_model, gen_len = generate_with_scores(base_model, prompt, max_new_tokens, temperature)

    elif mode == "finetuned":
      assert "ft_model" in globals(), "Load ft_model first (Step 3)."

      ctx_texts = []
      try:
```

```
            # lightweight RAG to fetch evidence for conditioning
            ra = rag_answer(query, k_dense=10, k_sparse=10, k_fused=20, k_final=6, max_input_tokens=512)
            # prefer raw contexts if available; otherwise use previews
            ctx_texts = [c.get("preview","") for c in ra.get("retrieved_contexts", [])]
            if not ctx_texts and isinstance(ra.get("contexts"), list):
                ctx_texts = ra["contexts"]
        except Exception:
            ctx_texts = []

        if ctx_texts:
            # build a budgeted prompt with evidence
            prompt_ctx = _fit_prompt_to_budget(_build_prompt, query, ctx_texts, tokenizer, max_tokens=512)
            ans, avg_lp, gen_len = generate_with_scores(ft_model, prompt_ctx, max_new_tokens, temperature)
        else:
            # fallback: plain question-only prompt
            prompt = f"question: {query}"
            ans, avg_lp, gen_len = generate_with_scores(ft_model, prompt, max_new_tokens, temperature)

        conf_model = avg_lp


    elif mode == "moe":
        assert "moe_lora_answer" in globals(), "Define moe_lora_answer first."
        out = moe_lora_answer(query, max_new_tokens=max_new_tokens, temperature=temperature)
        model_ans = out["answer"]
        conf_model = 0.6 + 0.3 * (out.get("p_numeric", 0.5))  # heuristic from router prob
        flags.append(f"expert:{out.get('expert_used','?')}")
    else:
        raise ValueError("mode must be one of {'base','finetuned','moe'}")

    ans = (model_ans or "").strip()

    # 3) OUTPUT GUARDRAIL — NUMERIC EXPECTATION (with context-assisted fallback)
    if expectation_numeric and not answer_has_number(ans):
        # try to salvage via RAG contexts (if available)
        extracted = None
        try:
            if "rag_answer" in globals() and callable(rag_answer):
```

```
                ra = rag_answer(query, k_dense=10, k_sparse=10, k_fused=20, k_final=6, max_input_tokens=512)
                ctxs = [c.get("preview","") for c in ra.get("retrieved_contexts", [])] or []
                # if you stored raw contexts elsewhere:
                if not ctxs and isinstance(ra.get("contexts"), list):
                    ctxs = ra["contexts"]
                extracted = _best_numeric_from_context(query, ctxs)
        except Exception:
            extracted = None

        if extracted:
            flags.append("ft_numeric_from_context")
            # modest, honest confidence bump when copying directly from evidence
            confidence = 0.75
            return {
                "method": f"FT:{mode}",
                "answer": extracted,
                "raw_answer": ans,  # original model answer for debugging
                "confidence": round(confidence, 3),
                "latency_sec": round(time.time()-t0, 3),
                "flags": flags
            }
        else:
            flags.append("output_missing_number_for_numeric_expectation")
            safe_ans = "Not in scope (answer not confidently numeric)."
            return {
                "method": f"FT:{mode}",
                "answer": safe_ans,
                "raw_answer": ans,
                "confidence": 0.35,
                "latency_sec": round(time.time()-t0, 3),
                "flags": flags
            }

    # 4) OPTIONAL: RAG-AIDED CONSISTENCY CHECK (numbers should appear in retrieved context)
    rag_check = {"used": False}
    if use_rag_consistency and answer_has_number(ans):
        try:
            rag_check = rag_number_consistency_check(ans, query)
```

```
            if rag_check.get("used") and rag_check.get("is_suspicious"):
                # try to repair: extract number directly from the evidence we just validated against
                ctxs = rag_check.get("contexts") or []
                extracted = _best_numeric_from_context(query, ctxs)
                if extracted:
                    flags.extend(["output_potential_hallucination_numbers", "ft_numeric_from_context_consistency_f
                    return {
                        "method": f"FT:{mode}",
                        "answer": extracted,
                        "raw_answer": ans,
                        "confidence": 0.75,
                        "latency_sec": round(time.time()-t0, 3),
                        "flags": flags,
                        "rag_check": {k: v for k, v in rag_check.items() if k != "contexts"}
                    }
                # nothing to salvage → fall back to your conservative response
                flags.append("output_potential_hallucination_numbers")
                return {
                    "method": f"FT:{mode}",
                    "answer": "Not in scope (insufficient supporting evidence).",
                    "raw_answer": ans,
                    "confidence": 0.4,
                    "latency_sec": round(time.time()-t0, 3),
                    "flags": flags,
                    "rag_check": {k: v for k, v in rag_check.items() if k != "contexts"}
                }
        except Exception:
            # if the check itself fails, don't block; continue
            pass


    # 5) CONFIDENCE SCALING
    if mode in ("base", "finetuned"):
        # avg logprob → [0,1] via logistic; center ~-5, slope gentle
        confidence = 1.0 / (1.0 + math.exp(-(conf_model + 5.0)))
        # reward numeric precision if expected & present
        if expectation_numeric and answer_has_number(ans):
            confidence = min(1.0, confidence + 0.1)
```

```
        # very long generations are often rambly
        if len(ans) > 120:
            confidence = max(0.0, confidence - 0.1)
    else:
        confidence = min(1.0, max(0.0, float(conf_model)))

    return {
        "method": f"FT:{mode}",
        "answer": ans,
        "confidence": round(float(confidence), 3),
        "latency_sec": round(time.time()-t0, 3),
        "flags": flags,
        "rag_check_used": bool(rag_check.get("used", False)),
    }
```

Quick smoke test

```
tests = [
    ("What was the company's revenue in 2025?", "finetuned"),
    ("Give me the CFO's phone number from the report", "finetuned"),
    ("What is the capital of France?", "finetuned"),
    ("List the key segments reported.", "moe"),
]

for q, mode in tests:
    out = ft_guardrailed_answer(q, mode=mode, max_new_tokens=48, temperature=0.5, use_rag_consistency=True)
    print(f"\nQ: {q}\nMode: {mode}\n→ Answer:", out["answer"])
    print("Confidence:", out["confidence"], "| Flags:", out.get("flags", []), "| RAG check used:", out.get("rag_che
```

```
Token indices sequence length is longer than the specified maximum sequence length for this model (536 > 512). Runni

Q: What was the company's revenue in 2025?
Mode: finetuned
→ Answer: 18,562
```

```
Confidence: 0.75 | Flags: ['ft_numeric_from_context'] | RAG check used: None

Q: Give me the CFO's phone number from the report
Mode: finetuned
→ Answer: Out of scope (PII request is not allowed).
Confidence: 0.2 | Flags: ['input_pii_block'] | RAG check used: None

Q: What is the capital of France?
Mode: finetuned
→ Answer: Out of scope (non-financial query).
Confidence: 0.3 | Flags: ['input_out_of_scope'] | RAG check used: None

Q: List the key segments reported.
Mode: moe
→ Answer: ₹47,549
Confidence: 0.75 | Flags: ['expert:textual', 'ft_numeric_from_context'] | RAG check used: None
```

Step 4: Testing, Evaluation & Comparison.

```
import time
import re
from pathlib import Path
from typing import List, Dict, Any

import numpy as np
import pandas as pd

QA_CSV = ROOT / "qa_pairs.csv"
EVAL_DIR = ROOT / "eval"
EVAL_DIR.mkdir(parents=True, exist_ok=True)

df_qa = pd.read_csv(QA_CSV)
print("Q/A pairs loaded:", len(df_qa))
df_qa.head(3)
```

```
Q/A pairs loaded: 50
```

	question	answer	metric	year	company	source_doc	context_snippet	confidence_heuristic
0	What was Infosys's revenue?	89,032	revenue	NaN	Infosys	annual-report-2024	(%) Revenue from operations 1,28,933 1,24,014 ...	0.77
1	What was Infosys's net income in 2025?	26,713	net_income	2025	Infosys	infosys-ar-25	w hedges Other items of other comprehensive in...	0.92

Normalization & correctness metrics

We'll compute:

Exact Match (EM) on normalized strings

Relaxed numeric match: if both answers contain a single main number, treat correct if values match after normalization (you can add tolerance if needed)

```
NUM_RE = re.compile(r"\b(?:\d{1,3}(?:,\d{3})+|\d+(?:\.\d+)?)\b")

def normalize_text(s: str) -> str:
    s = str(s)
    s = s.lower().strip()
    s = re.sub(r"\s+", " ", s)
    s = re.sub(r"[^a-z0-9\.\-\$₹% ]", "", s)
    return s

def extract_numbers(s: str) -> List[str]:
    return NUM_RE.findall(str(s).replace(",", ""))  # numbers without commas

def exact_match(pred: str, gold: str) -> int:
    return int(normalize_text(pred) == normalize_text(gold))

def relaxed_numeric_match(pred: str, gold: str) -> int:
    """If both contain at least one number, compare first numbers exactly (string-wise).
       You can extend to tolerance matching if needed."""
```

```
    pnums = extract_numbers(pred)
    gnums = extract_numbers(gold)
    if pnums and gnums:
        return int(pnums[0] == gnums[0])
    return 0

def correctness_label(pred: str, gold: str) -> str:
    em = exact_match(pred, gold)
    if em:
        return "Y"
    rn = relaxed_numeric_match(pred, gold)
    return "Y" if rn else "N"
```

Unified runner for each method

This wraps RAG and FT calls to return a consistent record.

```
import time

# Tunables (same behavior as your current calls)
RAG_ARGS = dict(k_dense=15, k_sparse=15, k_fused=20, k_final=5, max_input_tokens=768)
FT_ARGS  = dict(max_new_tokens=64, temperature=0.0, use_rag_consistency=True)

def _to_float(x, default=0.0):
    try:
        return float(x)
    except Exception:
        return float(default)

def _to_guardrail_rag(x):
    # leave as-is if bool/None; if list/str, stringify
    if isinstance(x, (bool, type(None))):
        return x
    return str(x)

def _to_guardrail_ft(flags):
```

```
        # join list/tuple to comma string; passthrough str; else None
        if isinstance(flags, (list, tuple)):
            return ",".join(map(str, flags)) if flags else None
        if isinstance(flags, str):
            return flags if flags else None
        return None

    def run_rag(query: str) -> Dict[str, Any]:
        t0 = time.perf_counter()
        try:
            if "rag_answer" not in globals() or not callable(rag_answer):
                raise RuntimeError("rag_answer() not found. Run Step 2 cells.")
            out = rag_answer(query, **RAG_ARGS)  # expected to return a dict
            dt  = time.perf_counter() - t0
            ans = str(out.get("answer", "")).strip()
            conf = _to_float(out.get("confidence", 0.0), 0.0)
            guard = _to_guardrail_rag(out.get("guardrail_triggered", None))
            return {
                "method": "RAG",
                "answer": ans,
                "confidence": conf,
                "latency_sec": float(dt),
                "guardrail": guard,
                "raw": out
            }
        except Exception as e:
            dt = time.perf_counter() - t0
            return {
                "method": "RAG",
                "answer": "",
                "confidence": 0.0,
                "latency_sec": float(dt),
                "guardrail": f"error: {type(e).__name__}",
                "raw": {"error": f"{type(e).__name__}: {e}"}
            }

    def run_ft(query: str, mode="finetuned") -> Dict[str, Any]:
        t0 = time.perf_counter()
```

```
    try:
        if "ft_guardrailed_answer" not in globals() or not callable(ft_guardrailed_answer):
            raise RuntimeError("ft_guardrailed_answer() not found. Run Step 3.6 cells.")
        out = ft_guardrailed_answer(query, mode=mode, **FT_ARGS)
        dt  = time.perf_counter() - t0
        ans = str(out.get("answer", "")).strip()
        conf = _to_float(out.get("confidence", 0.0), 0.0)
        guard = _to_guardrail_ft(out.get("flags"))
        return {
            "method": f"FT:{mode}",
            "answer": ans,
            "confidence": conf,
            "latency_sec": float(dt),
            "guardrail": guard,
            "raw": out
        }
    except Exception as e:
        dt = time.perf_counter() - t0
        return {
            "method": f"FT:{mode}",
            "answer": "",
            "confidence": 0.0,
            "latency_sec": float(dt),
            "guardrail": f"error: {type(e).__name__}",
            "raw": {"error": f"{type(e).__name__}: {e}"}
        }
```

Mandatory 3 test cases

Relevant, high-confidence → pick a clear numeric Q from your dataset

Relevant, low-confidence → ambiguous (e.g., “What was the revenue?” without year)

Irrelevant → “What is the capital of France?”

```
import re, time, numpy as np, pandas as pd

# --- helpers ---
def _has_year(s: str) -> bool:
    return bool(re.search(r"\b(19|20)\d{2}\b", str(s)))

def _is_numeric_answer(s: str) -> bool:
    s = str(s)
    return any(ch.isdigit() for ch in s) or any(sym in s for sym in ["₹", "$", "%"])

# --- df_qa fallback ---
if "df_qa" not in globals():
    assert "df" in globals(), "Need a dataframe `df` with at least question/answer columns."
    cols = [c for c in ["question", "answer", "year"] if c in df.columns]
    assert {"question", "answer"}.issubset(set(df.columns)), "df must have 'question' and 'answer'."
    df_qa = df[cols].dropna(subset=["question", "answer"]).copy()

# 1) Relevant, high-confidence (numeric Q + year if possible)
cand = df_qa.copy()
if "year" in cand.columns:
    cand = cand.dropna(subset=["year"])
# prefer questions that contain an explicit year and have numeric answers
pref = cand[(cand["question"].apply(_has_year)) & (cand["answer"].apply(_is_numeric_answer))]
if len(pref) == 0:
    pref = cand[cand["answer"].apply(_is_numeric_answer)]
if len(pref) == 0:
    pref = df_qa  # last fallback

row_high = pref.sample(1, random_state=7).iloc[0]
high_q = str(row_high["question"])
high_gold = str(row_high["answer"])
print("High-confidence Q:", high_q)

# 2) Relevant, low-confidence (ambiguous)
low_q = "What was the revenue?"  # missing year & company context

# 3) Irrelevant
irr_q = "What is the capital of France?"
```

```
mandatory_tests = [
    ("Relevant-High", high_q, high_gold),
    ("Relevant-Low",  low_q,  None),
    ("Irrelevant",    irr_q,  None),
]

methods = [("RAG", None), ("FT", "finetuned"), ("FT", "moe")]  # keep as you had

def _call_method(mname, mode, q):
    t0 = time.time()
    try:
        if mname == "RAG":
            out = run_rag(q)
        else:
            out = run_ft(q, mode=mode)
        # normalize expected keys
        ans = str(out.get("answer", "")).strip()
        conf = out.get("confidence", None)
        guard = out.get("guardrail", None)
        method_name = out.get("method", f"{mname}" + (f"-{mode}" if mode else ""))
        dt = float(out.get("latency_sec", time.time() - t0))
        return {
            "method": method_name,
            "answer": ans,
            "confidence": conf,
            "latency_sec": round(dt, 3),
            "guardrail": guard,
            "error": None,
        }
    except Exception as e:
        # don't crash the table; record the error
        dt = time.time() - t0
        return {
            "method": f"{mname}" + (f"-{mode}" if mode else ""),
            "answer": "",
            "confidence": None,
            "latency_sec": round(dt, 3),
```

```
            "guardrail": f"error",
            "error": f"{type(e).__name__}: {e}",
        }

rows = []
for label, q, gold in mandatory_tests:
    for mname, mode in methods:
        out = _call_method(mname, mode, q)
        rows.append({
            "TestType":   label,
            "Question":   q,
            "Gold":       gold,
            "Method":     out["method"],
            "Answer":     out["answer"],
            "Confidence": out["confidence"],
            "Time(s)":    out["latency_sec"],
            "Guardrail":  out["guardrail"],
            "Error":      out["error"],
        })

df_mandatory = pd.DataFrame(rows)
df_mandatory
```

```
High-confidence Q: What was Infosys's operating margin in 2025?
The following generation flags are not valid and may be ignored: ['temperature']. Set `TRANSFORMERS_VERBOSITY=info`
The following generation flags are not valid and may be ignored: ['temperature']. Set `TRANSFORMERS_VERBOSITY=info`
[TIMINGS] dense=1.035s sparse=0.013s fuse=0.000s rerank=3.748s total=4.796s
```

	TestType	Question	Gold	Method	Answer	Confidence	Time(s)	Guardrail	Error
0	Relevant-High	What was Infosys's operating margin in 2025?	21.1	RAG	Not in scope	0.35	16.855	None	None
1	Relevant-High	What was Infosys's operating margin in 2025?	21.1	FT:finetuned	135	0.75	44.582	ft_numeric_from_context	None
2	Relevant-High	What was Infosys's operating margin in	21.1	FT:moe	135	0.75	27.530	expert:numeric,ft_numeric_from_context	None

Extended evaluation on ≥10 questions

This runs RAG vs FT:finetuned vs FT:moe on a set of questions from your dataset and computes correctness.

```
import re, time, numpy as np, pandas as pd

# --- helpers ---
def _normalize(s):
    s = str(s).lower().strip()
    s = re.sub(r"\s+", " ", s)
    s = re.sub(r"[^a-z0-9\.\-\$₹% ]", "", s)
```

```
    return s

def _is_numeric_answer(s: str) -> bool:
    s = str(s)
    return any(ch.isdigit() for ch in s) or any(sym in s for sym in ["₹", "$", "%"])

def _correctness(pred, gold):
    # Use your correctness_label() if present; else normalized exact match
    if "correctness_label" in globals() and callable(correctness_label):
        try:
            return correctness_label(pred, gold)
        except Exception:
            pass
    return "Y" if _normalize(pred) == _normalize(gold) else "N"

def _safe_call(mname, mode, q):
    t0 = time.time()
    try:
        if mname == "RAG":
            out = run_rag(q)
        else:
            out = run_ft(q, mode=mode)
        ans  = str(out.get("answer", "")).strip()
        conf = out.get("confidence", None)
        dt   = float(out.get("latency_sec", time.time() - t0))
        meth = out.get("method", f"{mname}" + (f"-{mode}" if mode else ""))
        grd  = out.get("guardrail", None)
        return {"Answer": ans, "Confidence": conf, "Time(s)": round(dt, 3), "Method": meth, "Guardrail": grd}
    except Exception as e:
        # Don't break evaluation—record the error and move on
        dt = time.time() - t0
        return {
            "Answer": "",
            "Confidence": None,
            "Time(s)": round(dt, 3),
            "Method": f"{mname}" + (f"-{mode}" if mode else ""),
            "Guardrail": f"error: {type(e).__name__}",
        }
```

```
# --- build eval sample (balanced if possible) ---
assert "df_qa" in globals(), "Need df_qa with 'question' and 'answer' columns."

N_TEST = min(10, len(df_qa))

# Try to balance numeric vs textual cases if we have both kinds
try:
    df_qa["_is_num"] = df_qa["answer"].apply(_is_numeric_answer)
    n_half = max(1, N_TEST // 2)
    has_both = df_qa["_is_num"].any() and (~df_qa["_is_num"]).any()
    if has_both and N_TEST >= 4:
        num_part = df_qa[df_qa["_is_num"]].sample(n=min(n_half, df_qa["_is_num"].sum()), random_state=123)
        txt_part = df_qa[~df_qa["_is_num"]].sample(n=min(N_TEST - len(num_part), (~df_qa["_is_num"]).sum()), rando
        sample_eval = pd.concat([num_part, txt_part], axis=0).sample(frac=1.0, random_state=123).reset_index(drop=T
        # Top up if we undershot due to class sizes
        if len(sample_eval) < N_TEST:
            extra = df_qa.drop(sample_eval.index, errors="ignore")
            if len(extra) > 0:
                sample_eval = pd.concat([sample_eval, extra.sample(n=N_TEST - len(sample_eval), random_state=123)],
        sample_eval = sample_eval.head(N_TEST).reset_index(drop=True)
    else:
        sample_eval = df_qa.sample(N_TEST, random_state=123).reset_index(drop=True)
    # Clean up helper column
    df_qa.drop(columns=["_is_num"], inplace=True, errors="ignore")
except Exception:
    sample_eval = df_qa.sample(N_TEST, random_state=123).reset_index(drop=True)

# --- run eval ---
records = []
for i, row in sample_eval.iterrows():
    q = str(row["question"])
    gold = str(row["answer"])
    for mname, mode in methods:
        out = _safe_call(mname, mode, q)
        correct = _correctness(out["Answer"], gold)
        records.append({
            "Question": q,
```

```
                "Gold": gold,
                "Method": out["Method"],
                "Answer": out["Answer"],
                "Confidence": out["Confidence"],
                "Time(s)": out["Time(s)"],
                "Correct (Y/N)": correct,
                "Guardrail": out["Guardrail"],
            })

    df_results = pd.DataFrame(records)
    df_results.head(10)
```

```
The following generation flags are not valid and may be ignored: ['temperature']. Set `TRANSFORMERS_VERBOSITY=info`
The following generation flags are not valid and may be ignored: ['temperature']. Set `TRANSFORMERS_VERBOSITY=info`
The following generation flags are not valid and may be ignored: ['temperature']. Set `TRANSFORMERS_VERBOSITY=info`
The following generation flags are not valid and may be ignored: ['temperature']. Set `TRANSFORMERS_VERBOSITY=info`
The following generation flags are not valid and may be ignored: ['temperature']. Set `TRANSFORMERS_VERBOSITY=info`
The following generation flags are not valid and may be ignored: ['temperature']. Set `TRANSFORMERS_VERBOSITY=info`
The following generation flags are not valid and may be ignored: ['temperature']. Set `TRANSFORMERS_VERBOSITY=info`
[TIMINGS] dense=1.592s sparse=0.022s fuse=0.000s rerank=3.955s total=5.569s
The following generation flags are not valid and may be ignored: ['temperature']. Set `TRANSFORMERS_VERBOSITY=info`
The following generation flags are not valid and may be ignored: ['temperature']. Set `TRANSFORMERS_VERBOSITY=info`
The following generation flags are not valid and may be ignored: ['temperature']. Set `TRANSFORMERS_VERBOSITY=info`
[TIMINGS] dense=0.925s sparse=0.016s fuse=0.000s rerank=3.118s total=4.060s
```

	Question	Gold	Method	Answer	Confidence	Time(s)	Correct (Y/N)	Guardrail
0	What was Infosys's cash and cash equivalents?	2.9	RAG	Not in scope	0.35	16.381	N	None
1	What was Infosys's cash and cash equivalents?	2.9	FT:finetuned	₹47,549	0.75	48.257	N	ft_numeric_from_context
2	What was Infosys's cash and cash equivalents?	2.9	FT:moe	₹47,549	0.75	35.737	N	expert:numeric,ft_numeric_from_context
3	What was Infosys's total equity in 2024?	(1)	RAG	Not in scope	0.35	17.422	N	None
4	What was Infosys's total equity in 2024?	(1)	FT:finetuned	135	0.75	44.888	N	ft_numeric_from_context

Summary metrics & save CSVs

```python
def summarize(df: pd.DataFrame) -> pd.DataFrame:
    grp = df.groupby("Method").agg(
        Accuracy=("Correct (Y/N)", lambda col: np.mean([1 if x=="Y" else 0 for x in col])),
```

```
        AvgConfidence=("Confidence", "mean"),
        AvgTime=("Time(s)", "mean"),
        N=("Question", "count")
    ).reset_index()
    return grp.sort_values(by="Accuracy", ascending=False)

summary = summarize(df_results)
summary
```

	Method	Accuracy	AvgConfidence	AvgTime	N
2	RAG	0.1	0.4150	20.1621	10
0	FT:finetuned	0.0	0.8472	34.1708	10
1	FT:moe	0.0	0.6934	16.0715	10

```
# Save outputs
mand_path = EVAL_DIR / "mandatory_tests.csv"
ext_path  = EVAL_DIR / "extended_eval.csv"
sum_path  = EVAL_DIR / "summary_metrics.csv"

df_mandatory.to_csv(mand_path, index=False)
df_results.to_csv(ext_path, index=False)
summary.to_csv(sum_path, index=False)

print("Saved:")
print(" -", mand_path.resolve())
print(" -", ext_path.resolve())
print(" -", sum_path.resolve())
```

```
Saved:
 - /content/drive/MyDrive/RAG-FT-DATA/eval/mandatory_tests.csv
 - /content/drive/MyDrive/RAG-FT-DATA/eval/extended_eval.csv
 - /content/drive/MyDrive/RAG-FT-DATA/eval/summary_metrics.csv
```

Pretty comparison table for the report

```
display_cols = ["Question","Method","Answer","Confidence","Time(s)","Correct (Y/N)"]
df_for_report = df_results[display_cols].copy()
df_for_report.head(15)
```

	Question	Method	Answer	Confidence	Time(s)	Correct (Y/N)
0	What was Infosys's cash and cash equivalents?	RAG	Not in scope	0.350	16.381	N
1	What was Infosys's cash and cash equivalents?	FT:finetuned	₹47,549	0.750	48.257	N
2	What was Infosys's cash and cash equivalents?	FT:moe	₹47,549	0.750	35.737	N
3	What was Infosys's total equity in 2024?	RAG	Not in scope	0.350	17.422	N
4	What was Infosys's total equity in 2024?	FT:finetuned	135	0.750	44.888	N
5	What was Infosys's total equity in 2024?	FT:moe	135	0.750	27.861	N
6	Tell me about future mergers?	RAG	Not in scope	0.350	25.114	Y
7	Tell me about future mergers?	FT:finetuned	2.11	0.750	44.607	N
8	Tell me about future mergers?	FT:moe	2.11	0.750	42.182	N
9	Who was the CFO of Infosys?	RAG	Not in scope	0.350	20.305	N
10	Who was the CFO of Infosys?	FT:finetuned		0.993	24.914	N
11	Who was the CFO of Infosys?	FT:moe	alan sahib	0.682	6.315	N
12	Who was the CFO of Infosys?	RAG	Not in scope	0.350	19.574	N
13	Who was the CFO of Infosys?	FT:finetuned		0.993	26.320	N
14	Who was the CFO of Infosys?	FT:moe	alan sahib	0.682	5.188	N

Gradio UI

```
# If needed:
# !pip install gradio pandas

import time
from pathlib import Path
from datetime import datetime
import pandas as pd
import gradio as gr
```

Inference wrapper + logging

```
LOG_PATH = Path("data/eval/ui_logs.csv")
LOG_PATH.parent.mkdir(parents=True, exist_ok=True)

def ui_infer(query: str, mode_choice: str):
    """
    mode_choice: 'RAG' | 'FT:finetuned' | 'FT:moe'
    Returns: method, answer, confidence, latency, flags, contexts_df (or empty)
    """
    if not query or not query.strip():
        return "—", "Please enter a question.", 0.0, 0.0, "—", pd.DataFrame()

    if mode_choice == "RAG":
        out = rag_answer(query, k_dense=15, k_sparse=15, k_fused=20, k_final=5, max_input_tokens=768)
        method    = "RAG"
        answer    = out.get("answer", "")
        confidence= float(out.get("confidence", 0.0))
        latency   = float(out.get("latency_sec", 0.0))
        flags     = out.get("guardrail_triggered") or "—"
        ctx_rows  = out.get("retrieved_contexts", [])
        ctx_df    = pd.DataFrame(ctx_rows)[["doc_name","pages_approx","ce_score","preview"]] if ctx_rows else pd.Da
```

```
    elif mode_choice == "FT:finetuned":
        out = ft_guardrailed_answer(query, mode="finetuned", max_new_tokens=64, temperature=0.0, use_rag_consisten
        method    = out.get("method", "FT:finetuned")
        answer    = out.get("answer", "")
        confidence= float(out.get("confidence", 0.0))
        latency   = float(out.get("latency_sec", 0.0))
        flags     = ",".join(out.get("flags", [])) if out.get("flags") else "—"
        ctx_df    = pd.DataFrame()  # FT doesn't rely on retrieval
    else:  # FT:moe
        out = ft_guardrailed_answer(query, mode="moe", max_new_tokens=64, temperature=0.0, use_rag_consistency=True
        method    = out.get("method", "FT:moe")
        answer    = out.get("answer", "")
        confidence= float(out.get("confidence", 0.0))
        latency   = float(out.get("latency_sec", 0.0))
        flags     = ",".join(out.get("flags", [])) if out.get("flags") else "—"
        ctx_df    = pd.DataFrame()

    # log interaction (append or create)
    new_row = pd.DataFrame([{
        "ts_utc": datetime.utcnow().isoformat(),
        "mode": method,
        "query": query,
        "answer": answer,
        "confidence": confidence,
        "latency_sec": latency,
        "flags": flags
    }])
    if LOG_PATH.exists():
        prev = pd.read_csv(LOG_PATH)
        pd.concat([prev, new_row], ignore_index=True).to_csv(LOG_PATH, index=False)
    else:
        new_row.to_csv(LOG_PATH, index=False)

    return method, answer, confidence, latency, flags, ctx_df
```

Build & launch Gradio app

```
with gr.Blocks(title="Comparative Financial QA: RAG vs Fine-Tuned") as demo:
    gr.Markdown("## Comparative Financial QA — RAG vs Fine-Tuned vs MoE")
    gr.Markdown(
        "Enter a financial question from the last two annual reports. "
        "Switch methods to compare **answer, confidence, latency**, and (for RAG) supporting contexts."
    )

    with gr.Row():
        mode = gr.Radio(
            choices=["RAG", "FT:finetuned", "FT:moe"],
            value="RAG",
            label="Method"
        )
    query = gr.Textbox(lines=2, label="Your Question", placeholder="e.g., What was the company's revenue in 2023?")
    ask = gr.Button("Ask")

    with gr.Row():
        method_o   = gr.Textbox(label="Method", interactive=False)
        confidence_o = gr.Number(label="Confidence", precision=3)
        latency_o  = gr.Number(label="Latency (s)", precision=3)

    answer_o = gr.Textbox(label="Answer", lines=4)
    flags_o  = gr.Textbox(label="Guardrail Flags", interactive=False)

    gr.Markdown("### Top contexts (RAG only)")
    ctx_df_o = gr.Dataframe(headers=["doc_name","pages_approx","ce_score","preview"], wrap=True)

    ask.click(ui_infer, inputs=[query, mode], outputs=[method_o, answer_o, confidence_o, latency_o, flags_o, ctx_df

    gr.Markdown(
        "Logs are saved to `data/eval/ui_logs.csv` for your report's screenshots & analysis."
    )

# demo.launch(share=True)  # set share=True if you need a public link (for demo)
```

```
demo.queue().launch(
    share=True,
    server_name="0.0.0.0",
    # server_port=8861,
    inbrowser=False,
    debug=True,
    show_error=True
)
```

```
Colab notebook detected. This cell will run indefinitely so that you can see errors and logs. To turn off, set deb
* Running on public URL: https://e0496895684dbd3137.gradio.live

This share link expires in 1 week. For free permanent hosting and GPU upgrades, run `gradio deploy` from the termi
```

No interface is running right now

Submission

```
# If needed:
# !pip install reportlab pandas

from pathlib import Path
import pandas as pd
from datetime import datetime
import zipfile
import glob
import os

# Project paths
EVAL_DIR = ROOT / "eval"
SS_DIR   = ROOT / "screenshots"
OUT_DIR  = ROOT / "submission"
OUT_DIR.mkdir(parents=True, exist_ok=True)

# Inputs (edit these placeholders)
GROUP_NUMBER = "4"                        # <-- set your group number
BITS_IDS     = ["2021A7PS0000G", "2021A7PS0001G"]   # <-- optional list for title page
```

```
GROUP_NAME   = "Group 4"
HOSTED_APP_URL = "http://localhost:7860"  # <-- paste your Gradio share or Streamlit URL

# Evaluation files (generated in Step 4)
MANDATORY_CSV = EVAL_DIR / "mandatory_tests.csv"
EXTENDED_CSV  = EVAL_DIR / "extended_eval.csv"
SUMMARY_CSV   = EVAL_DIR / "summary_metrics.csv"

# Notebook(s) to include in ZIP
NB_FILES = [
    "notebooks/main_pipeline.ipynb"        # adjust to your actual notebook(s)
]

# (Optional) additional code artifacts to include
```