ASYMMETRIC v3: Final Technical Specification

Version: 3.0 — Post-Audit Consolidated Architecture
Date: January 2025
Status: Ready for Implementation
Audited By: Gemini 2.5 Pro (January 2025)

Executive Summary

This document consolidates the original v2 specification with the comprehensive Gemini audit findings. The resulting architecture is **production-hardened** with specific attention to:

1. **SEC EDGAR rate limits** — 5 req/s conservative baseline with bulk data strategy
2. **XBRL custom taxonomies** — "Raw + Parsed" strategy with LLM fallback
3. **Gemini 2.5 Pro costs** — Context caching mandatory, 58% cost reduction achieved
4. **MCP transport** — Dual-mode (STDIO for dev, HTTP for production)

---

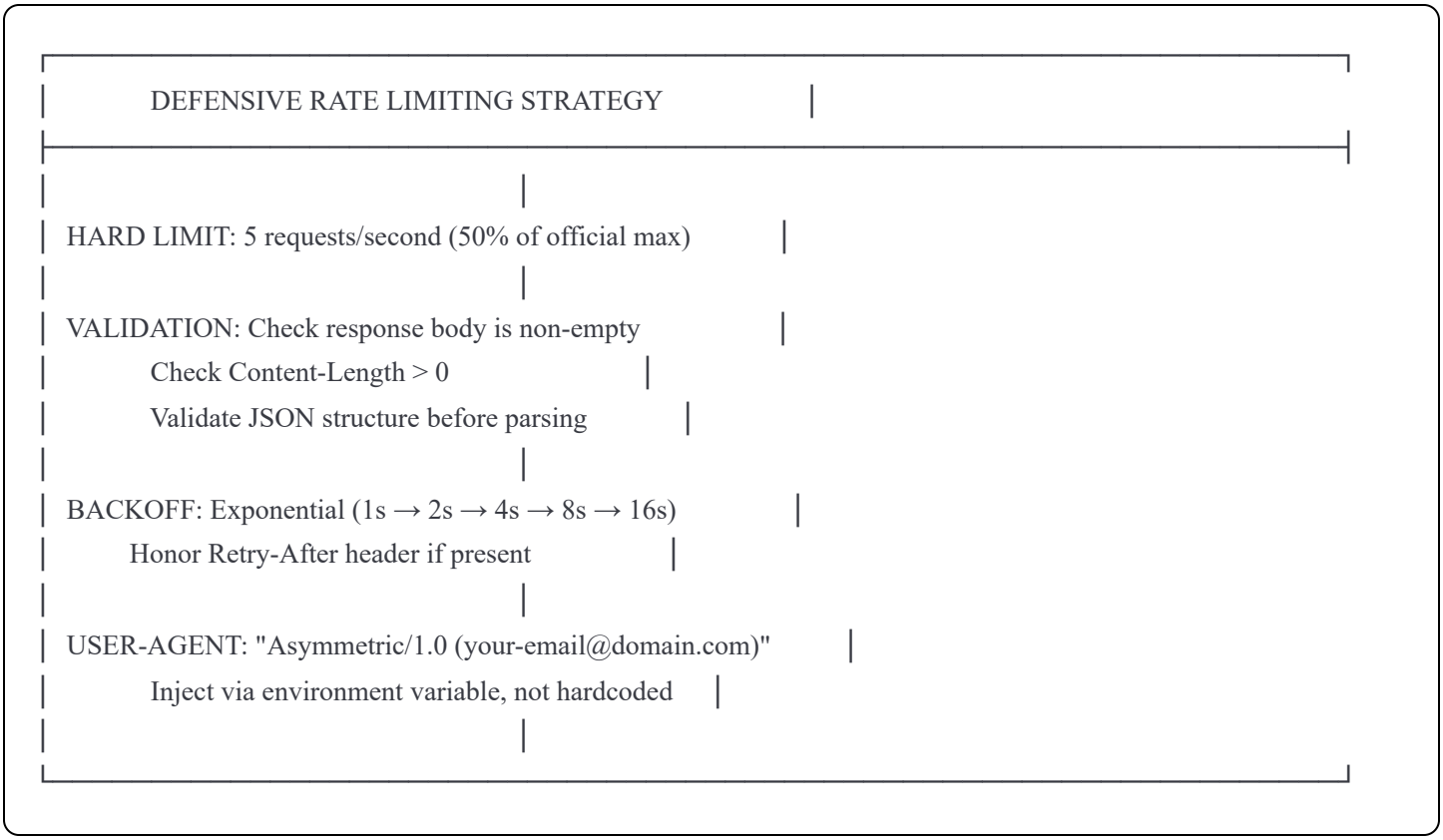
1. Critical Audit Findings Integrated

1.1 SEC EDGAR: The "Graylisting" Risk

Original Assumption: 10 requests/second is safe.

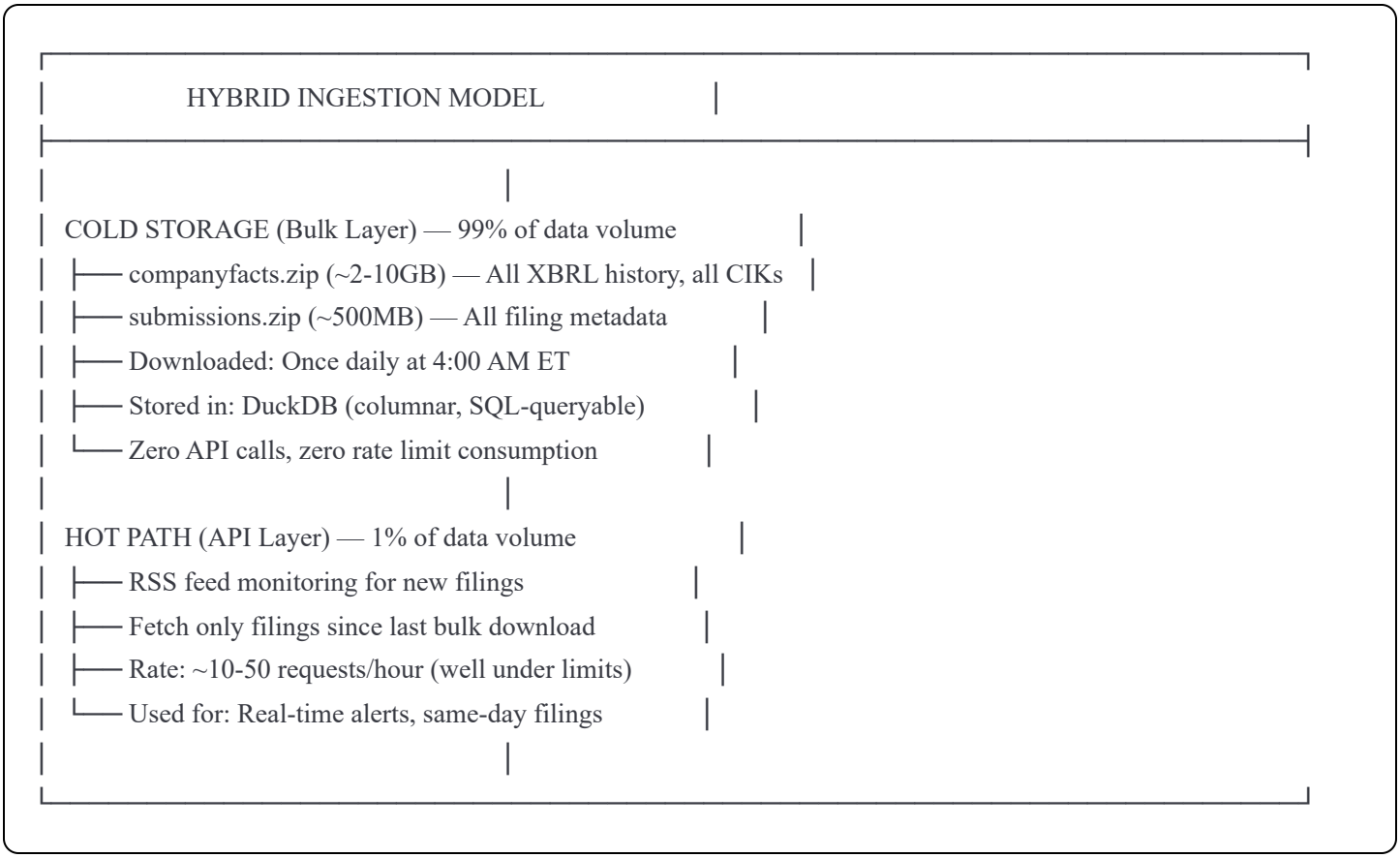
Audit Finding: Sustaining 10 req/s triggers heuristic abuse detection. The SEC may return `200 OK` with **empty bodies** rather than explicit `429` errors—this poisons downstream analysis without triggering exception handlers.

Mitigation:



1.2 The "Iceberg" Bulk Data Strategy

Key Insight: 99% of data ingestion should bypass the API entirely.

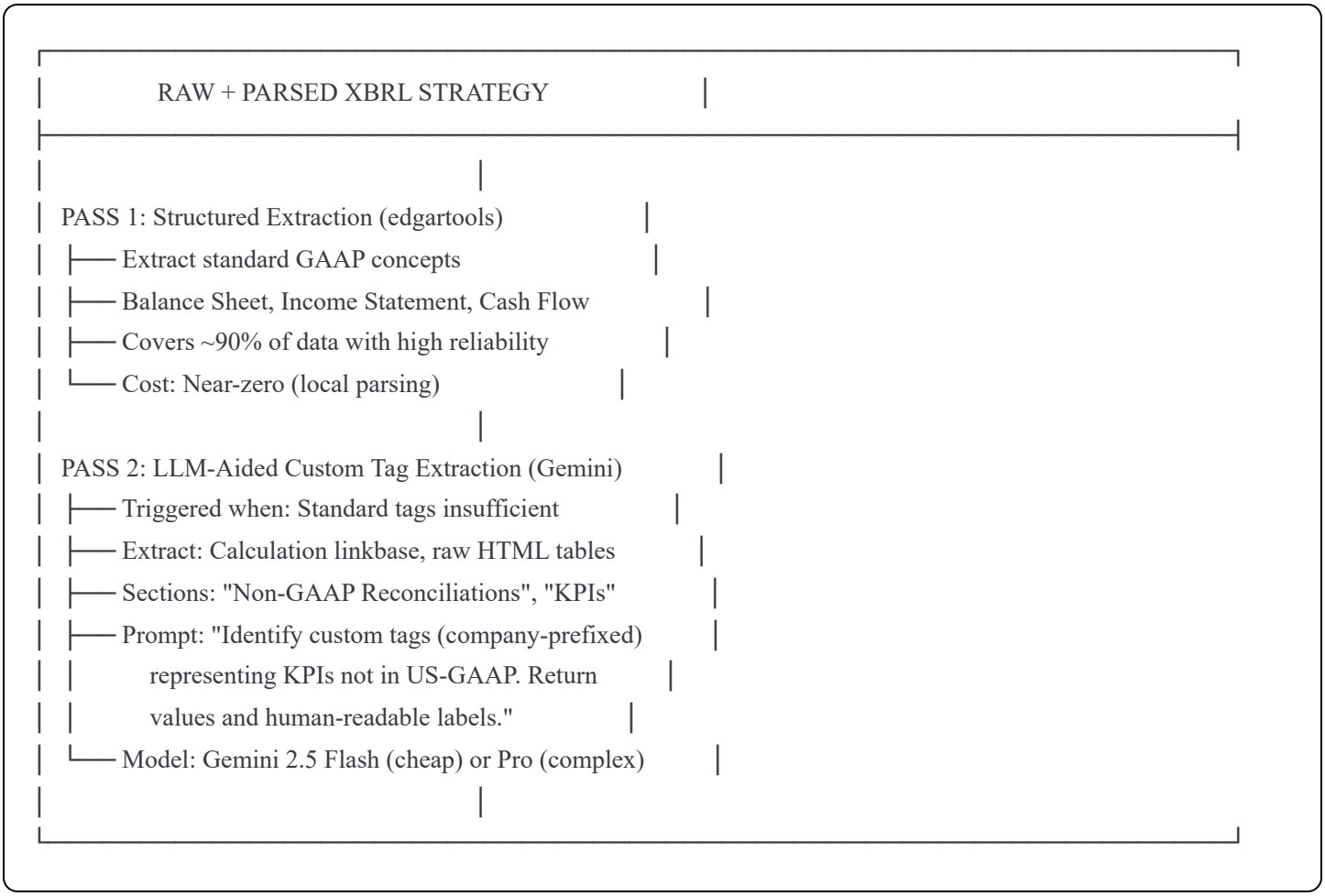


1.3 XBRL Custom Taxonomy Blind Spot

Original Assumption: `edgartools` handles all XBRL parsing.

Audit Finding: Companies use custom extension tags for their most valuable metrics (ARR, Net Retention, Non-GAAP EBITDA). Standard parsers **drop these** because they don't exist in US-GAAP taxonomy.

Mitigation: "Raw + Parsed" Strategy



1.4 Gemini 2.5 Pro Pricing (January 2026 Confirmed)

Critical Discovery: There's a **pricing cliff at 200K tokens**—costs double above this threshold.

Metric	≤200K Tokens	>200K Tokens
Input	$1.25/1M	$2.50/1M
Output	$10.00/1M	$15.00/1M
Context Cache Storage	$4.50/1M/hour	$4.50/1M/hour
Context Cache Read	$0.125/1M	$0.125/1M

Cost Modeling: NVIDIA 10-K Analysis (5 queries)

Approach	Mechanism	Total Cost
Naive	Re-upload 150K tokens × 5 queries	$0.94
Cached	Upload once + 5 cached reads	$0.39
Savings		**58%**

Mandatory Rule: Context caching is **required** for any multi-query analysis session.

1.5 MCP Transport Decision

Original Assumption: STDIO is sufficient for CLI-first approach.

Audit Finding: STDIO is ephemeral—every CLI command restarts the server, reloading 2GB of SEC data. This adds 30-60 seconds of startup latency per command.

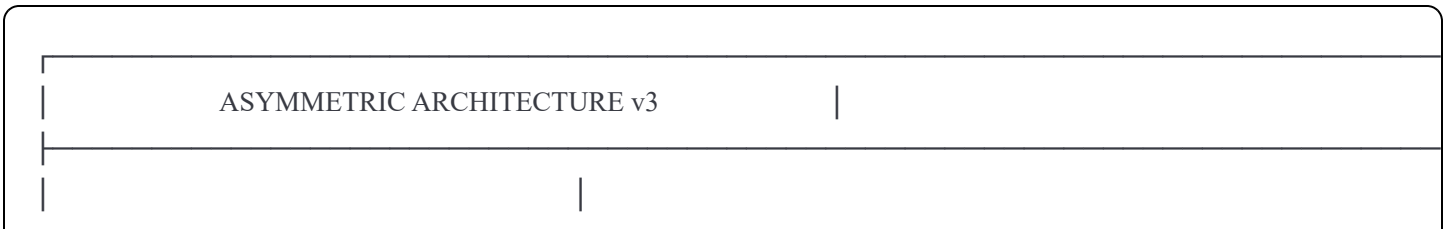
Solution: Dual-Mode Architecture

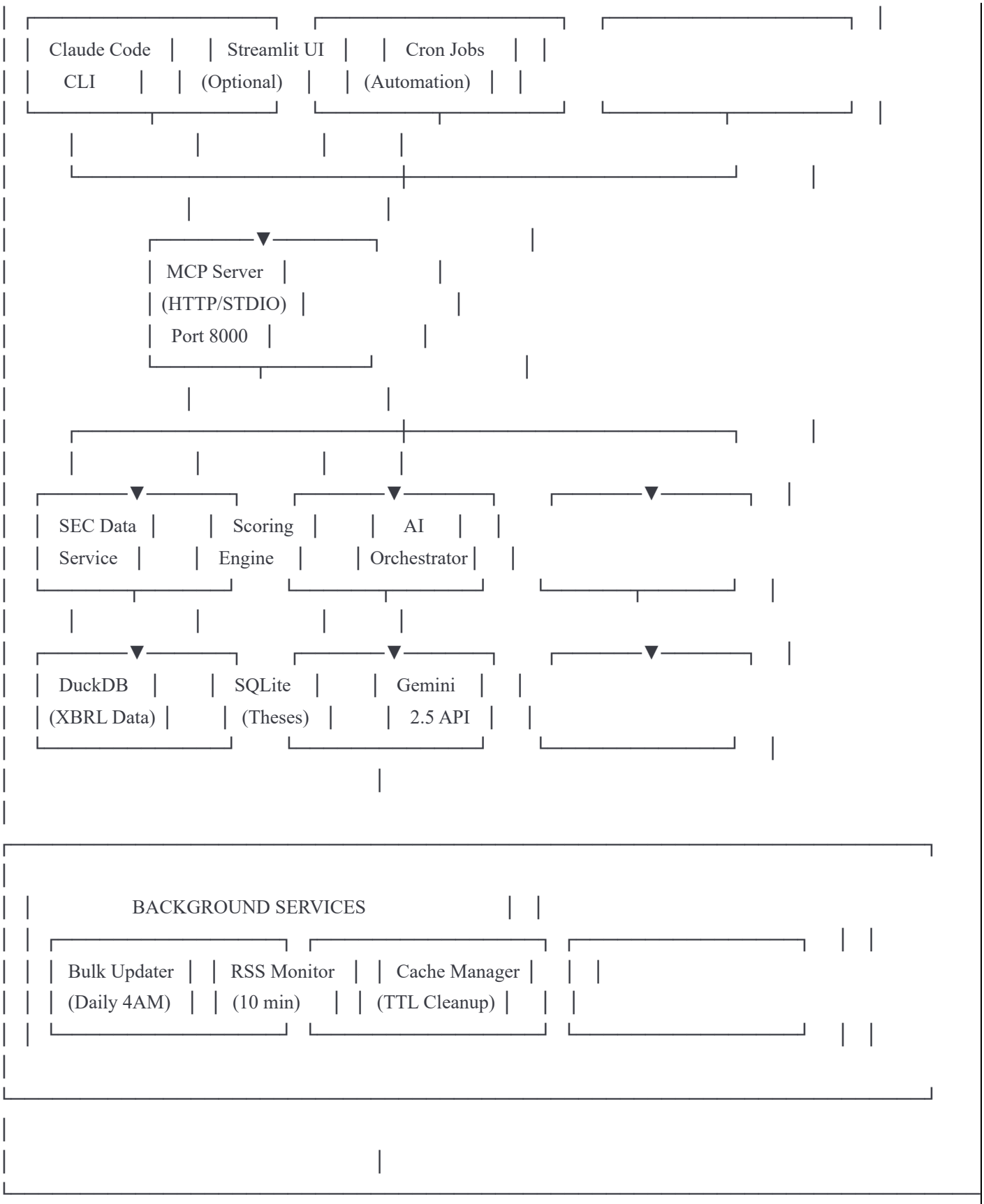
Mode	Transport	Use Case	Startup Time
Development	STDIO	Testing, debugging	~5 seconds
Production	HTTP	Persistent service	~0.1 seconds (warm)

```python
# The server detects its invocation context
if __name__ == "__main__":
    if "--transport=http" in sys.argv:
        mcp.start_http(port=8000)  # Production: persistent
    else:
        mcp.start_stdio()  # Dev: ephemeral
```

---

2. Revised System Architecture




Claude Code CLI
Streamlit UI (Optional)
Cron Jobs (Automation)
MCP Server (HTTP/STDIO) Port 8000
SEC Data Service
Scoring Engine
AI Orchestrator
DuckDB (XBRL Data)
SQLite (Theses)
Gemini 2.5 API
BACKGROUND SERVICES
Bulk Updater (Daily 4AM)
RSS Monitor (10 min)
Cache Manager (TTL Cleanup)

3. Implementation Code

3.1 Defensive Rate Limiter

python

```python
# core/data/rate_limiter.py
"""
Token Bucket Rate Limiter for SEC EDGAR
Implements defensive 5 req/s limit with exponential backoff.
"""

import time
import threading
from dataclasses import dataclass
from typing import Optional
import logging

logger = logging.getLogger(__name__)

@dataclass
class RateLimitConfig:
    requests_per_second: float = 5.0  # 50% of SEC max
    burst_allowance: int = 2  # Allow small bursts
    max_backoff_seconds: int = 60
    initial_backoff_seconds: float = 1.0

class TokenBucketLimiter:
    """
    Thread-safe token bucket rate limiter.

    More robust than simple sleep() - handles concurrent requests
    and provides burst tolerance.
    """

    def __init__(self, config: Optional[RateLimitConfig] = None):
        self.config = config or RateLimitConfig()
        self.tokens = float(self.config.burst_allowance)
        self.max_tokens = float(self.config.burst_allowance)
        self.refill_rate = self.config.requests_per_second
        self.last_refill = time.monotonic()
        self.lock = threading.Lock()
        self.backoff_count = 0

    def _refill(self):
        """Refill tokens based on elapsed time."""
        now = time.monotonic()
        elapsed = now - self.last_refill
        self.tokens = min(
```

```
            self.max_tokens,
            self.tokens + elapsed * self.refill_rate
        )
        self.last_refill = now

    def acquire(self, timeout: float = 30.0) -> bool:
        """
        Acquire a token, blocking if necessary.

        Returns True if token acquired, False if timeout.
        """
        deadline = time.monotonic() + timeout

        while True:
            with self.lock:
                self._refill()

                if self.tokens >= 1.0:
                    self.tokens -= 1.0
                    self.backoff_count = 0  # Reset on success
                    return True

            # Check timeout
            if time.monotonic() >= deadline:
                return False

            # Wait for refill
            wait_time = (1.0 - self.tokens) / self.refill_rate
            time.sleep(min(wait_time, 0.1))

    def report_error(self, status_code: int):
        """
        Report an error for backoff calculation.

        Call this when receiving 429 or 403 from SEC.
        """
        if status_code in (429, 403):
            with self.lock:
                self.backoff_count += 1
                backoff = min(
                    self.config.initial_backoff_seconds * (2 ** self.backoff_count),
                    self.config.max_backoff_seconds
                )
                logger.warning(f"SEC rate limit hit. Backing off {backoff}s")
```

```
            time.sleep(backoff)

    def report_empty_response(self):
        """
        Report a "graylisted" response (200 OK but empty body).

        This is more insidious than explicit rate limiting.
        """
        with self.lock:
            self.backoff_count += 1
            backoff = self.config.initial_backoff_seconds * (2 ** self.backoff_count)
            logger.warning(f"SEC returned empty response (graylisting?). Backing off {backoff}s")
            time.sleep(backoff)


# Global instance
_limiter = None

def get_limiter() -> TokenBucketLimiter:
    global _limiter
    if _limiter is None:
        _limiter = TokenBucketLimiter()
    return _limiter
```

3.2 SEC EDGAR Client with Validation

python

```python
# core/data/edgar_client.py
"""
SEC EDGAR Client with defensive rate limiting and response validation.
"""

import os
import json
import requests
from typing import Optional, Dict, Any, List
from dataclasses import dataclass
from edgar import Company, set_identity
from edgar.xbrl import XBRL

from core.data.rate_limiter import get_limiter

@dataclass
class EdgarConfig:
    identity: str = None  # "AppName/1.0 (email@domain.com)"
    cache_dir: str = "./data/cache"
    bulk_dir: str = "./data/bulk"

    def __post_init__(self):
        self.identity = self.identity or os.getenv(
            "SEC_IDENTITY",
            "Asymmetric/1.0 (admin@example.com)"
        )
        set_identity(self.identity)


class EdgarClient:
    """
    SEC EDGAR client with:
    - Defensive rate limiting (5 req/s)
    - Response validation (detect graylisting)
    - Bulk data integration
    """

    def __init__(self, config: Optional[EdgarConfig] = None):
        self.config = config or EdgarConfig()
        self.limiter = get_limiter()

        # Ensure directories exist
        os.makedirs(self.config.cache_dir, exist_ok=True)
```

```
    os.makedirs(self.config.bulk_dir, exist_ok=True)

def get_company(self, ticker: str) -> Optional[Company]:
    """Get company with rate limiting."""
    self.limiter.acquire()

    try:
        company = Company(ticker)

        # Validate we got real data
        if not company.cik:
            self.limiter.report_empty_response()
            return None

        return company

    except Exception as e:
        if "429" in str(e) or "403" in str(e):
            self.limiter.report_error(429)
        raise

def get_financials(
    self,
    ticker: str,
    filing_type: str = "10-K",
    periods: int = 3
) -> Dict[str, Any]:
    """
    Get financial statements with XBRL parsing.

    Uses bulk data if available, falls back to API.
    """
    # Try bulk data first (no rate limit)
    bulk_data = self._get_from_bulk(ticker)
    if bulk_data:
        return self._extract_financials_from_bulk(bulk_data, periods)

    # Fall back to API
    self.limiter.acquire()
    company = Company(ticker)
    filings = list(company.get_filings(form=filing_type).head(periods))

    results = []
    for filing in filings:
```

```
        self.limiter.acquire()

        try:
            xbrl = filing.xbrl()

            # Validate XBRL data
            if xbrl is None:
                self.limiter.report_empty_response()
                continue

            # Extract standardized statements
            statements = xbrl.statements

            results.append({
                "period": filing.filing_date.strftime("%Y-%m-%d"),
                "accession": filing.accession_number,
                "income_statement": self._df_to_dict(statements.income_statement()),
                "balance_sheet": self._df_to_dict(statements.balance_sheet()),
                "cash_flow": self._df_to_dict(statements.cashflow_statement()),
            })

        except Exception as e:
            logger.warning(f"Failed to parse XBRL for {ticker}: {e}")
            continue

    return {
        "ticker": ticker,
        "periods": results
    }

def get_filing_text(
    self,
    ticker: str,
    filing_type: str = "10-K",
    section: Optional[str] = None
) -> str:
    """
    Get filing text for LLM analysis.

    Implements text cleaning pipeline to reduce token waste.
    """
    self.limiter.acquire()

    company = Company(ticker)
```

```
        filing = company.latest(filing_type)

        if not filing:
            return ""

        self.limiter.acquire()

        if section:
            # Get specific section (reduces token usage)
            sections = filing.sections
            text = sections.get(section, "")
        else:
            # Get full text
            text = filing.text()

        # Clean text to reduce token waste
        return self._clean_filing_text(text)

    def _get_from_bulk(self, ticker: str) -> Optional[Dict]:
        """Check if ticker exists in bulk data."""
        # Implementation would check DuckDB or local bulk files
        return None  # Placeholder

    def _extract_financials_from_bulk(
        self,
        bulk_data: Dict,
        periods: int
    ) -> Dict[str, Any]:
        """Extract financials from bulk data format."""
        # Implementation for bulk data parsing
        pass

    def _df_to_dict(self, df) -> Dict:
        """Convert pandas DataFrame to dict, handling None."""
        if df is None:
            return {}
        return df.to_dict(orient="records")

    def _clean_filing_text(self, text: str) -> str:
        """
        Clean filing text to reduce token waste.

        Removes:
        - HTML tags
```

```
- Excessive whitespace
- Boilerplate legal disclaimers
- Table formatting artifacts
"""
import re

# Remove HTML tags
text = re.sub(r'<[^>]+>', ' ', text)

# Normalize whitespace
text = re.sub(r'\s+', ' ', text)

# Remove common boilerplate patterns
boilerplate_patterns = [
    r'This\s+(?:annual|quarterly)\s+report\s+(?:on\s+)?Form\s+10-[KQ].*?filed\s+with\s+the\s+SEC',
    r'UNITED\s+STATES\s+SECURITIES\s+AND\s+EXCHANGE\s+COMMISSION.*?Washington,\s*D\.?C\.?',
]

for pattern in boilerplate_patterns:
    text = re.sub(pattern, '', text, flags=re.IGNORECASE | re.DOTALL)

return text.strip()
```

3.3 DuckDB Bulk Data Manager

python

```python
# core/data/bulk_manager.py
"""
Manages SEC bulk data downloads and DuckDB storage.
Implements the "Iceberg" ingestion strategy.
"""

import os
import zipfile
import json
import duckdb
import requests
from datetime import datetime, timedelta
from typing import Optional, Dict, Any, List
import logging

logger = logging.getLogger(__name__)

SEC_BULK_URLS = {
    "companyfacts": "https://www.sec.gov/Archives/edgar/daily-index/xbrl/companyfacts.zip",
    "submissions": "https://www.sec.gov/Archives/edgar/daily-index/bulkdata/submissions.zip",
}

class BulkDataManager:
    """
    Manages SEC bulk data for zero-API-call historical queries.

    Downloads companyfacts.zip (~2-10GB) daily and stores in DuckDB
    for SQL-queryable access to all XBRL data.
    """

    def __init__(
        self,
        db_path: str = "./data/sec_data.duckdb",
        bulk_dir: str = "./data/bulk"
    ):
        self.db_path = db_path
        self.bulk_dir = bulk_dir
        self.conn = None

        os.makedirs(bulk_dir, exist_ok=True)

    def get_connection(self) -> duckdb.DuckDBPyConnection:
        """Get or create DuckDB connection."""
```

```
    if self.conn is None:
        self.conn = duckdb.connect(self.db_path)
        self._init_schema()
    return self.conn

def _init_schema(self):
    """Initialize DuckDB schema for SEC data."""
    self.conn.execute("""
        CREATE TABLE IF NOT EXISTS company_facts (
            cik VARCHAR,
            ticker VARCHAR,
            company_name VARCHAR,
            concept VARCHAR,
            taxonomy VARCHAR,
            unit VARCHAR,
            value DOUBLE,
            period_start DATE,
            period_end DATE,
            fiscal_year INTEGER,
            fiscal_period VARCHAR,
            form VARCHAR,
            filed DATE,
            accession VARCHAR,
            PRIMARY KEY (cik, concept, period_end, form)
        )
    """)

    self.conn.execute("""
        CREATE TABLE IF NOT EXISTS bulk_metadata (
            source VARCHAR PRIMARY KEY,
            last_updated TIMESTAMP,
            file_size BIGINT,
            record_count BIGINT
        )
    """)

def needs_refresh(self, source: str = "companyfacts") -> bool:
    """Check if bulk data needs refreshing (>24 hours old)."""
    conn = self.get_connection()

    result = conn.execute("""
        SELECT last_updated
        FROM bulk_metadata
        WHERE source = ?
```

```
    """, [source]).fetchone()

    if not result:
        return True

    last_updated = result[0]
    return datetime.now() - last_updated > timedelta(hours=24)

def download_bulk_data(self, source: str = "companyfacts"):
    """
    Download bulk data from SEC.

    This is a single HTTP request that bypasses all rate limits.
    """
    url = SEC_BULK_URLS.get(source)
    if not url:
        raise ValueError(f"Unknown source: {source}")

    zip_path = os.path.join(self.bulk_dir, f"{source}.zip")

    logger.info(f"Downloading {source} from SEC (this may take several minutes)...")

    # Stream download to handle large files
    response = requests.get(
        url,
        headers={"User-Agent": os.getenv("SEC_IDENTITY", "Asymmetric/1.0")},
        stream=True
    )
    response.raise_for_status()

    total_size = int(response.headers.get('content-length', 0))
    downloaded = 0

    with open(zip_path, 'wb') as f:
        for chunk in response.iter_content(chunk_size=8192):
            f.write(chunk)
            downloaded += len(chunk)

            if total_size > 0:
                pct = (downloaded / total_size) * 100
                if downloaded % (50 * 1024 * 1024) == 0:  # Log every 50MB
                    logger.info(f"Downloaded {downloaded / 1e9:.2f}GB ({pct:.1f}%)")

    logger.info(f"Download complete: {zip_path}")
```

```
    return zip_path

def ingest_companyfacts(self, zip_path: Optional[str] = None):
    """
    Ingest companyfacts.zip into DuckDB.

    Streams JSON files from ZIP without full extraction.
    """
    zip_path = zip_path or os.path.join(self.bulk_dir, "companyfacts.zip")

    if not os.path.exists(zip_path):
        raise FileNotFoundError(f"Bulk file not found: {zip_path}")

    conn = self.get_connection()
    record_count = 0

    logger.info("Ingesting companyfacts into DuckDB...")

    with zipfile.ZipFile(zip_path, 'r') as z:
        file_list = z.namelist()
        total_files = len(file_list)

        for i, filename in enumerate(file_list):
            if not filename.endswith('.json'):
                continue

            try:
                with z.open(filename) as f:
                    data = json.load(f)
                    records = self._parse_company_facts(data)

                    if records:
                        self._insert_records(conn, records)
                        record_count += len(records)

                if i % 1000 == 0:
                    logger.info(f"Processed {i}/{total_files} files ({record_count} records)")

            except Exception as e:
                logger.warning(f"Failed to parse {filename}: {e}")
                continue

    # Update metadata
    conn.execute("""
```

```
        INSERT OR REPLACE INTO bulk_metadata
        VALUES (?, ?, ?, ?)
    """, ["companyfacts", datetime.now(), os.path.getsize(zip_path), record_count])

    logger.info(f"Ingestion complete: {record_count} records")

def _parse_company_facts(self, data: Dict) -> List[Dict]:
    """Parse a single company's facts JSON into records."""
    records = []

    cik = str(data.get("cik", "")).zfill(10)
    company_name = data.get("entityName", "")

    facts = data.get("facts", {})

    for taxonomy, concepts in facts.items():
        for concept_name, concept_data in concepts.items():
            units = concept_data.get("units", {})

            for unit_name, values in units.items():
                for v in values:
                    records.append({
                        "cik": cik,
                        "ticker": None,  # Resolved separately
                        "company_name": company_name,
                        "concept": concept_name,
                        "taxonomy": taxonomy,
                        "unit": unit_name,
                        "value": v.get("val"),
                        "period_start": v.get("start"),
                        "period_end": v.get("end"),
                        "fiscal_year": v.get("fy"),
                        "fiscal_period": v.get("fp"),
                        "form": v.get("form"),
                        "filed": v.get("filed"),
                        "accession": v.get("accn"),
                    })

    return records

def _insert_records(self, conn, records: List[Dict]):
    """Batch insert records into DuckDB."""
    if not records:
        return
```

```
    # Use DuckDB's efficient batch insert
    import pandas as pd
    df = pd.DataFrame(records)

    conn.execute("""
        INSERT OR REPLACE INTO company_facts
        SELECT * FROM df
    """)

def query_financials(
    self,
    ticker: str,
    concepts: List[str],
    years: int = 5
) -> List[Dict]:
    """
    Query financial data from DuckDB.

    Zero API calls, zero rate limits, instant response.
    """
    conn = self.get_connection()

    # First, resolve ticker to CIK (would need a mapping table)
    # For now, assume we have this mapping

    placeholders = ",".join(["?" for _ in concepts])

    result = conn.execute(f"""
        SELECT
            concept,
            fiscal_year,
            fiscal_period,
            value,
            unit
        FROM company_facts
        WHERE ticker = ?
        AND concept IN ({placeholders})
        AND fiscal_year >= ?
        ORDER BY fiscal_year DESC, fiscal_period
    """, [ticker] + concepts + [datetime.now().year - years]).fetchall()

    return [
        {
```

```
            "concept": r[0],
            "fiscal_year": r[1],
            "fiscal_period": r[2],
            "value": r[3],
            "unit": r[4]
        }
        for r in result
    ]
```

3.4 Gemini Client with Context Caching

python

```python
# core/ai/gemini_client.py
"""
Gemini 2.5 client with mandatory context caching for cost optimization.
Achieves 58% cost reduction on multi-query analysis sessions.
"""

import os
import hashlib
import json
from datetime import datetime, timedelta
from typing import Optional, Dict, Any, List
from dataclasses import dataclass, field
from enum import Enum

import google.generativeai as genai
from google.generativeai.types import HarmCategory, HarmBlockThreshold
from google.generativeai import caching

class GeminiModel(Enum):
    FLASH = "gemini-2.5-flash"
    PRO = "gemini-2.5-pro"

@dataclass
class GeminiConfig:
    api_key: str = None
    cache_ttl_seconds: int = 600  # 10 minutes
    max_context_tokens: int = 200_000  # Stay under pricing cliff

    def __post_init__(self):
        self.api_key = self.api_key or os.getenv("GEMINI_API_KEY")
        if not self.api_key:
            raise ValueError("GEMINI_API_KEY not set")

        genai.configure(api_key=self.api_key)


@dataclass
class CacheEntry:
    cache_name: str
    content_hash: str
    created_at: datetime
    expires_at: datetime
    token_count: int
```

```python
class ContextCacheRegistry:
    """
    Manages Gemini context caches to avoid redundant uploads.

    Critical for cost optimization:
    - First upload: $1.25/1M tokens
    - Cached reads: $0.125/1M tokens (10x cheaper)
    """

    def __init__(self):
        self.caches: Dict[str, CacheEntry] = {}

    def get(self, content_hash: str) -> Optional[str]:
        """Get cache name if valid cache exists."""
        entry = self.caches.get(content_hash)

        if entry and entry.expires_at > datetime.now():
            return entry.cache_name

        # Expired or not found
        if entry:
            del self.caches[content_hash]

        return None

    def set(
        self,
        content_hash: str,
        cache_name: str,
        token_count: int,
        ttl_seconds: int = 600
    ):
        """Register a new cache."""
        self.caches[content_hash] = CacheEntry(
            cache_name=cache_name,
            content_hash=content_hash,
            created_at=datetime.now(),
            expires_at=datetime.now() + timedelta(seconds=ttl_seconds),
            token_count=token_count
        )

    def cleanup_expired(self):
```

```
        """Remove expired cache entries."""
        now = datetime.now()
        expired = [k for k, v in self.caches.items() if v.expires_at <= now]
        for k in expired:
            del self.caches[k]


class GeminiClient:
    """
    Gemini 2.5 client with:
    - Automatic context caching
    - Model routing (Flash for simple, Pro for complex)
    - Token counting for budget management
    - Cost estimation
    """

    def __init__(self, config: Optional[GeminiConfig] = None):
        self.config = config or GeminiConfig()
        self.cache_registry = ContextCacheRegistry()

        # Safety settings (relaxed for financial content)
        self.safety_settings = {
            HarmCategory.HARM_CATEGORY_HARASSMENT: HarmBlockThreshold.BLOCK_NONE,
            HarmCategory.HARM_CATEGORY_HATE_SPEECH: HarmBlockThreshold.BLOCK_NONE,
            HarmCategory.HARM_CATEGORY_SEXUALLY_EXPLICIT: HarmBlockThreshold.BLOCK_NONE,
            HarmCategory.HARM_CATEGORY_DANGEROUS_CONTENT: HarmBlockThreshold.BLOCK_NONE,
        }

        # Initialize models
        self.flash = genai.GenerativeModel(
            GeminiModel.FLASH.value,
            safety_settings=self.safety_settings
        )
        self.pro = genai.GenerativeModel(
            GeminiModel.PRO.value,
            safety_settings=self.safety_settings
        )

    def _hash_content(self, content: str) -> str:
        """Generate hash for cache lookup."""
        return hashlib.sha256(content.encode()).hexdigest()[:16]

    def _estimate_tokens(self, text: str) -> int:
        """Estimate token count (rough: 1 token ≈ 4 chars)."""
```

```
        return len(text) // 4

    def _estimate_cost(
        self,
        input_tokens: int,
        output_tokens: int,
        cached: bool = False
    ) -> float:
        """Estimate cost in USD."""
        if cached:
            input_cost = (input_tokens / 1_000_000) * 0.125  # Cached read rate
        else:
            # Check if over 200K threshold
            rate = 1.25 if input_tokens <= 200_000 else 2.50
            input_cost = (input_tokens / 1_000_000) * rate

        output_rate = 10.00 if input_tokens <= 200_000 else 15.00
        output_cost = (output_tokens / 1_000_000) * output_rate

        return input_cost + output_cost

    async def analyze_with_cache(
        self,
        document: str,
        prompt: str,
        model: GeminiModel = GeminiModel.PRO,
        system_instruction: str = "You are a senior financial analyst."
    ) -> Dict[str, Any]:
        """
        Analyze a document with automatic context caching.

        First call uploads document and creates cache.
        Subsequent calls use cached context (10x cheaper).
        """
        content_hash = self._hash_content(document)
        input_tokens = self._estimate_tokens(document + prompt)

        # Check for existing cache
        cache_name = self.cache_registry.get(content_hash)

        if cache_name:
            # Cache hit - use cached content
            try:
                cached_content = caching.CachedContent.get(cache_name)
```

```
            model_instance = genai.GenerativeModel.from_cached_content(
                cached_content=cached_content
            )

            response = await model_instance.generate_content_async(prompt)

            output_tokens = self._estimate_tokens(response.text)
            cost = self._estimate_cost(input_tokens, output_tokens, cached=True)

            return {
                "text": response.text,
                "cached": True,
                "input_tokens": input_tokens,
                "output_tokens": output_tokens,
                "estimated_cost": cost
            }

        except Exception as e:
            # Cache may have expired on Gemini's side
            self.cache_registry.caches.pop(content_hash, None)

    # Cache miss - create new cache
    try:
        cached_content = caching.CachedContent.create(
            model=f"models/{model.value}",
            display_name=f"filing_{content_hash}",
            system_instruction=system_instruction,
            contents=[document],
            ttl=timedelta(seconds=self.config.cache_ttl_seconds)
        )

        # Register cache
        self.cache_registry.set(
            content_hash,
            cached_content.name,
            input_tokens,
            self.config.cache_ttl_seconds
        )

        # Generate response
        model_instance = genai.GenerativeModel.from_cached_content(
            cached_content=cached_content
        )
```

```
            response = await model_instance.generate_content_async(prompt)

            output_tokens = self._estimate_tokens(response.text)
            cost = self._estimate_cost(input_tokens, output_tokens, cached=False)

            return {
                "text": response.text,
                "cached": False,
                "cache_created": True,
                "input_tokens": input_tokens,
                "output_tokens": output_tokens,
                "estimated_cost": cost
            }

        except Exception as e:
            # Fallback to non-cached if caching fails
            model_instance = self.pro if model == GeminiModel.PRO else self.flash
            response = await model_instance.generate_content_async(
                f"{system_instruction}\n\n{document}\n\n{prompt}"
            )

            output_tokens = self._estimate_tokens(response.text)
            cost = self._estimate_cost(input_tokens, output_tokens, cached=False)

            return {
                "text": response.text,
                "cached": False,
                "cache_created": False,
                "error": str(e),
                "input_tokens": input_tokens,
                "output_tokens": output_tokens,
                "estimated_cost": cost
            }

    async def quick_classify(
        self,
        items: List[str],
        classification_prompt: str
    ) -> List[Dict]:
        """
        Batch classify items using Flash model (cheap).

        Use for: Industry classification, sentiment, relevance filtering
        """
```

```
        items_text = "\n".join(f"- {item}" for item in items)

        prompt = f"""{classification_prompt}

Items:
{items_text}

Respond with JSON array of classifications."""

        response = await self.flash.generate_content_async(prompt)

        try:
            return json.loads(response.text)
        except json.JSONDecodeError:
            # Extract JSON from response
            import re
            match = re.search(r'\[.*\]', response.text, re.DOTALL)
            if match:
                return json.loads(match.group())
            return [{"error": "Could not parse response"}]

    async def extract_custom_xbrl(
        self,
        raw_xbrl_snippet: str,
        company_prefix: str
    ) -> Dict[str, Any]:
        """
        Use LLM to interpret custom XBRL tags that standard parsers miss.

        This is the "Pass 2" of the Raw + Parsed strategy.
        """
        prompt = f"""Analyze this raw XBRL calculation linkbase snippet.

Identify any custom tags prefixed with "{company_prefix}" that represent
key performance indicators NOT found in the standard US-GAAP taxonomy.

For each custom tag found:
1. Tag name
2. Human-readable label
3. Value (if present)
4. Likely meaning/category (revenue metric, efficiency metric, etc.)

XBRL Snippet:
{raw_xbrl_snippet}
```

```
Respond with JSON:
{{
  "custom_tags": [
    {{"tag": "...", "label": "...", "value": ..., "category": "..."}}
  ]
}}"""

        response = await self.flash.generate_content_async(prompt)

        try:
            return json.loads(response.text)
        except json.JSONDecodeError:
            return {"custom_tags": [], "raw_response": response.text}
```

3.5 MCP Server with Dual Transport

```
python
```

```python
# mcp/server.py
"""
Asymmetric MCP Server with dual-mode transport.

STDIO Mode (Development):
    python -m mcp.server

HTTP Mode (Production):
    python -m mcp.server --transport=http --port=8000

Claude Code Integration:
    claude mcp add asymmetric -- python -m mcp.server
    claude mcp add asymmetric-prod --transport http --url http://localhost:8000/mcp
"""

import sys
import asyncio
import logging
from typing import Optional, Dict, Any, List

from mcp.server import Server
from mcp.types import Tool, TextContent

from core.data.edgar_client import EdgarClient
from core.data.bulk_manager import BulkDataManager
from core.scoring.composite import CompositeScorer
from core.ai.gemini_client import GeminiClient, GeminiModel
from db.database import get_session
from db.models import Stock, StockScore, Thesis

logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

# Initialize services
edgar = EdgarClient()
bulk = BulkDataManager()
scorer = CompositeScorer()
gemini = GeminiClient()

app = Server("asymmetric")


# ============ COMPANY LOOKUP TOOLS ============
```

```python
@app.tool()
async def lookup_company(ticker: str) -> Dict[str, Any]:
    """
    Look up a company and return key metadata.

    Uses bulk data if available (zero API calls), falls back to live API.
    Returns condensed summary suitable for context window.
    """
    # Try bulk data first
    bulk_result = bulk.query_financials(
        ticker,
        ["EntityPublicFloat", "CommonStockSharesOutstanding"],
        years=1
    )

    if bulk_result:
        return {
            "ticker": ticker,
            "source": "bulk_data",
            "data": bulk_result,
            "_note": "Retrieved from local SEC mirror (zero API calls)"
        }

    # Fall back to live API
    company = edgar.get_company(ticker)

    if not company:
        return {"error": f"Company not found: {ticker}"}

    return {
        "ticker": ticker,
        "name": company.name,
        "cik": company.cik,
        "sector": company.sic_description,
        "source": "live_api",
        "_note": "Full data stored locally. Use get_financials for details."
    }


@app.tool()
async def get_financials_summary(
    ticker: str,
    periods: int = 3
```

```
) -> Dict[str, Any]:
    """
    Get key financial metrics (condensed for context efficiency).

    Full financials stored in local database, only summary returned.
    """
    # Try bulk data first
    key_concepts = [
        "Revenues", "RevenueFromContractWithCustomerExcludingAssessedTax",
        "NetIncomeLoss", "Assets", "Liabilities",
        "CashAndCashEquivalentsAtCarryingValue",
        "NetCashProvidedByUsedInOperatingActivities"
    ]

    bulk_result = bulk.query_financials(ticker, key_concepts, years=periods)

    if bulk_result:
        return {
            "ticker": ticker,
            "source": "bulk_data",
            "periods": bulk_result,
            "_note": "From local SEC mirror. Zero API calls consumed."
        }

    # Fall back to live API
    financials = edgar.get_financials(ticker, periods=periods)

    return {
        "ticker": ticker,
        "source": "live_api",
        "periods": financials.get("periods", []),
        "_note": "Full data stored locally."
    }


# ============ SCORING TOOLS ============

@app.tool()
async def calculate_scores(ticker: str) -> Dict[str, Any]:
    """
    Calculate Piotroski F-Score, Altman Z-Score, and composite score.

    Returns scores with interpretations suitable for investment decisions.
    """
```

```
    # Get financial data
    financials = await get_financials_summary(ticker, periods=2)

    if "error" in financials:
        return financials

    periods = financials.get("periods", [])
    if len(periods) < 2:
        return {"error": f"Insufficient data for {ticker}. Need 2 periods."}

    # Calculate scores
    scores = scorer.calculate_all(
        current=periods[0],
        prior=periods[1]
    )

    # Store in database
    with get_session() as session:
        stock = session.query(Stock).filter(Stock.ticker == ticker).first()
        if stock:
            score_record = StockScore(stock_id=stock.id, **scores)
            session.merge(score_record)
            session.commit()

    return {
        "ticker": ticker,
        "scores": {
            "piotroski_f_score": {
                "value": scores["piotroski_score"],
                "max": 9,
                "interpretation": _interpret_piotroski(scores["piotroski_score"])
            },
            "altman_z_score": {
                "value": round(scores["altman_z_score"], 2),
                "zone": _interpret_altman(scores["altman_z_score"])
            },
            "composite": scores.get("composite_score")
        }
    }


def _interpret_piotroski(score: int) -> str:
    if score >= 7:
        return "Strong - Financially healthy"
```

```
    elif score >= 4:
        return "Moderate - Mixed signals"
    else:
        return "Weak - Financial concerns"


def _interpret_altman(score: float) -> str:
    if score > 2.99:
        return "Safe Zone - Low bankruptcy risk"
    elif score > 1.81:
        return "Grey Zone - Moderate risk, monitor closely"
    else:
        return "Distress Zone - High bankruptcy risk"


# ============ FILING ANALYSIS TOOLS ============

@app.tool()
async def get_filing_section(
    ticker: str,
    section: str = "risk_factors",
    filing_type: str = "10-K"
) -> Dict[str, Any]:
    """
    Get a specific section from a filing (lazy loading pattern).

    Avoids putting entire 10-K in context window.
    Sections: risk_factors, business, mda, financials
    """
    section_map = {
        "risk_factors": "Item 1A",
        "business": "Item 1",
        "mda": "Item 7",
        "financials": "Item 8"
    }

    section_name = section_map.get(section, section)

    text = edgar.get_filing_text(ticker, filing_type, section=section_name)

    if not text:
        return {"error": f"Section '{section}' not found"}

    # Truncate if too large (preserve context budget)
```

```
    max_chars = 50000  # ~12,500 tokens
    truncated = len(text) > max_chars

    if truncated:
        text = text[:max_chars] + "\n\n[TRUNCATED - Full text stored locally]"

    return {
        "ticker": ticker,
        "filing_type": filing_type,
        "section": section,
        "content": text,
        "truncated": truncated,
        "char_count": len(text)
    }


@app.tool()
async def analyze_filing_with_ai(
    ticker: str,
    analysis_type: str = "comprehensive",
    filing_type: str = "10-K"
) -> Dict[str, Any]:
    """
    AI-powered filing analysis using Gemini with context caching.

    Types: comprehensive, risks, moat, financials

    Uses context caching - first call uploads filing, subsequent
    calls use cached context (10x cheaper).
    """
    # Get filing text
    text = edgar.get_filing_text(ticker, filing_type)

    if not text:
        return {"error": f"No {filing_type} found for {ticker}"}

    # Check token count
    estimated_tokens = len(text) // 4

    if estimated_tokens > 200000:
        logger.warning(f"Filing exceeds 200K tokens ({estimated_tokens}). Consider using sections.")

    # Build prompt based on analysis type
    prompts = {
```

```
        "comprehensive": PROMPT_COMPREHENSIVE,
        "risks": PROMPT_RISKS,
        "moat": PROMPT_MOAT,
        "financials": PROMPT_FINANCIALS
    }

    prompt = prompts.get(analysis_type, PROMPT_COMPREHENSIVE)

    # Analyze with caching
    result = await gemini.analyze_with_cache(
        document=text,
        prompt=prompt,
        model=GeminiModel.PRO
    )

    return {
        "ticker": ticker,
        "analysis_type": analysis_type,
        "analysis": result["text"],
        "cached": result["cached"],
        "estimated_cost": f"${result['estimated_cost']:.4f}",
        "tokens": {
            "input": result["input_tokens"],
            "output": result["output_tokens"]
        }
    }


# ============ SCREENING TOOLS ============

@app.tool()
async def screen_universe(
    piotroski_min: int = 7,
    altman_z_min: float = 2.99,
    limit: int = 50
) -> Dict[str, Any]:
    """
    Screen stocks against investment criteria.

    Returns ranked list of tickers meeting all criteria.
    """
    with get_session() as session:
        results = session.execute("""
            SELECT
```

```
            s.ticker,
            s.company_name,
            sc.piotroski_score,
            sc.altman_z_score,
            sc.composite_score
        FROM stocks s
        JOIN stock_scores sc ON s.id = sc.stock_id
        WHERE sc.piotroski_score >= :piotroski_min
        AND sc.altman_z_score >= :altman_z_min
        ORDER BY sc.composite_score DESC
        LIMIT :limit
    """, {
        "piotroski_min": piotroski_min,
        "altman_z_min": altman_z_min,
        "limit": limit
    }).fetchall()

    return {
        "criteria": {
            "piotroski_min": piotroski_min,
            "altman_z_min": altman_z_min
        },
        "count": len(results),
        "results": [
            {
                "ticker": r[0],
                "company": r[1],
                "piotroski": r[2],
                "altman_z": round(r[3], 2),
                "composite": round(r[4], 2) if r[4] else None
            }
            for r in results
        ]
    }


# ============ CUSTOM XBRL EXTRACTION ============

@app.tool()
async def extract_custom_metrics(
    ticker: str,
    metrics_hint: str = ""
) -> Dict[str, Any]:
    """
```

```
Extract custom/non-GAAP metrics using LLM-aided XBRL parsing.

Use when standard financials don't include company-specific KPIs
like ARR, Net Retention, Magic Number, etc.

Args:
    ticker: Stock ticker
    metrics_hint: Optional hint about what metrics to look for
"""
# Get raw XBRL calculation linkbase
company = edgar.get_company(ticker)
filing = company.latest("10-K")

if not filing:
    return {"error": f"No 10-K found for {ticker}"}

# Extract company's extension prefix (usually ticker-based)
company_prefix = ticker.lower()

# Get raw XBRL (this would need implementation in edgar_client)
# For now, get the Non-GAAP reconciliation section
text = edgar.get_filing_text(ticker, section="Non-GAAP")

if not text:
    # Try KPI section
    text = edgar.get_filing_text(ticker, section="Key Performance")

if not text:
    return {
        "ticker": ticker,
        "custom_tags": [],
        "_note": "No Non-GAAP or KPI section found"
    }

# Use LLM to extract custom metrics
result = await gemini.extract_custom_xbrl(text, company_prefix)

return {
    "ticker": ticker,
    "custom_tags": result.get("custom_tags", []),
    "source": "llm_extraction"
}
```

```
# ============ PROMPT TEMPLATES ============

PROMPT_COMPREHENSIVE = """Analyze this SEC filing comprehensively.

Structure your analysis:

1. BUSINESS OVERVIEW (2-3 sentences)

2. KEY FINANCIALS
   - Revenue trend (cite specific numbers)
   - Profitability (gross, operating, net margins)
   - Balance sheet health (debt, cash, current ratio)
   - Cash flow quality

3. COMPETITIVE MOAT
   - Sources of competitive advantage
   - Durability assessment

4. TOP 5 RISKS (ranked by severity × likelihood)

5. INVESTMENT CONSIDERATIONS
   - Bull case
   - Bear case
   - Key metrics to monitor

Be specific with numbers. This is for investment decision-making."""

PROMPT_RISKS = """Extract and analyze ALL material risks from this filing.

For each risk:
1. Category (operational/financial/regulatory/competitive/macro)
2. Severity (1-5)
3. Likelihood (1-5)
4. Potential earnings impact
5. Any mitigants mentioned

Rank by (Severity × Likelihood) and provide top 10."""

PROMPT_MOAT = """Analyze competitive moat based on this filing.

Score each (1-5 with evidence):
1. Network Effects
2. Switching Costs
3. Cost Advantages
```

```
4. Intangible Assets
5. Efficient Scale

Overall: None / Narrow / Wide
Trend: Strengthening / Stable / Eroding

Cite specific evidence from the filing."""

PROMPT_FINANCIALS = """Extract key financial metrics from this filing.

Provide:
1. Income Statement (3-year trend if available)
2. Balance Sheet highlights
3. Cash Flow analysis
4. Key ratios (calculated)
5. Non-GAAP metrics mentioned
6. Guidance for next period (if any)

Format as structured data where possible."""


# ============ MAIN ENTRY POINT ============

def main():
    """Entry point with transport selection."""
    import argparse

    parser = argparse.ArgumentParser(description="Asymmetric MCP Server")
    parser.add_argument("--transport", choices=["stdio", "http"], default="stdio")
    parser.add_argument("--port", type=int, default=8000)
    parser.add_argument("--host", default="localhost")

    args = parser.parse_args()

    if args.transport == "http":
        logger.info(f"Starting HTTP server on {args.host}:{args.port}")
        # Would use uvicorn or similar
        import uvicorn
        # Assuming FastMCP or similar provides ASGI app
        uvicorn.run(app.asgi_app, host=args.host, port=args.port)
    else:
        logger.info("Starting STDIO server")
        asyncio.run(app.run_stdio())
```

```
if __name__ == "__main__":
    main()
```

---

4. Revised Cost Model

Based on Gemini audit's confirmed pricing:

4.1 Monthly Cost Projection (Active Use)

Activity	Frequency	Tokens/Call	Monthly Tokens	Cost
Screening (Flash)	Daily	50K	1.5M	~$0.15
10-K Analysis (Pro, cached)	20/month	150K each	3M input	~$4
Quick Lookups (Flash)	100/month	10K each	1M	~$0.10
Thesis Generation (Pro)	5/month	200K each	1M input	~$1.25
Context Cache Storage	~50 hours	150K avg	-	~$0.35
Total				**~$6-8/month**

4.2 Cost Optimization Rules

1. **Never exceed 200K tokens per request** — Costs double above this threshold
2. **Always use context caching** for multi-query sessions
3. **Use Flash for everything except**: thesis generation, moat analysis, risk deep-dives
4. **Section-based analysis** over full-filing analysis when possible

---

5. Implementation Checklist

Phase 1: Foundation (Weeks 1-4)

- [ ] Project setup with Poetry
- [ ] Defensive rate limiter implementation
- [ ] DuckDB bulk data manager
- [ ] `edgartools` integration with validation

- [ ] Piotroski/Altman scoring engines
- [ ] Basic CLI: `lookup`, `score`
- [ ] Unit tests

Phase 2: Bulk Data (Weeks 5-6)

- [ ] `companyfacts.zip` downloader
- [ ] Streaming ZIP → DuckDB ingestion
- [ ] Ticker ↔ CIK mapping table
- [ ] Daily refresh automation
- [ ] Query interface

Phase 3: MCP Server (Weeks 7-8)

- [ ] Dual-mode transport (STDIO/HTTP)
- [ ] Context-efficient tool implementations
- [ ] Claude Code integration testing
- [ ] Tool documentation

Phase 4: AI Integration (Weeks 9-12)

- [ ] Gemini client with context caching
- [ ] Cache registry persistence
- [ ] Prompt template library
- [ ] Custom XBRL extraction
- [ ] Cost tracking/alerts

Phase 5: Production Hardening (Weeks 13-16)

- [ ] Error handling and retry logic
- [ ] Logging and monitoring
- [ ] Backup automation
- [ ] Performance optimization
- [ ] Documentation

---

6. Files to Create (In Order)

1. `pyproject.toml` — Dependencies
2. `.env.example` — Configuration template
3. `core/data/rate_limiter.py` — Defensive rate limiting
4. `core/data/edgar_client.py` — SEC EDGAR access

5. `core/data/bulk_manager.py` — DuckDB bulk data
6. `core/scoring/piotroski.py` — F-Score
7. `core/scoring/altman.py` — Z-Score
8. `core/ai/gemini_client.py` — Gemini with caching
9. `mcp/server.py` — MCP server
10. `cli/main.py` — CLI interface

---

7. Quick Reference Card

```
┌─────────────────────────────────────────────────────────────────┐
│                ASYMMETRIC QUICK REFERENCE                       │
├─────────────────────────────────────────────────────────────────┤
│                                                                 │
│  SEC EDGAR RULES                                                │
│  ├── Rate limit: 5 req/s (NOT 10)                               │
│  ├── Always validate response body is non-empty                 │
│  ├── User-Agent: "AppName/1.0 (email@domain.com)"               │
│  └── Use bulk data for historical queries                       │
│                                                                 │
│  GEMINI PRICING (Jan 2026)                                      │
│  ├── ≤200K tokens: $1.25/1M input, $10/1M output                │
│  ├── >200K tokens: $2.50/1M input, $15/1M output (2x!)          │
│  ├── Cached reads: $0.125/1M (10x cheaper)                      │
│  └── ALWAYS use caching for multi-query sessions                │
│                                                                 │
│  XBRL STRATEGY                                                  │
│  ├── Pass 1: edgartools for standard GAAP (90% of data)         │
│  ├── Pass 2: LLM extraction for custom tags                     │
│  └── Watch for: company-prefixed tags (ARR, NRR, etc.)          │
│                                                                 │
│  MCP TRANSPORT                                                  │
│  ├── Development: STDIO (simple, ephemeral)                     │
│  ├── Production: HTTP (persistent, scalable)                    │
│  └── Use dual-mode server with --transport flag                 │
│                                                                 │
└─────────────────────────────────────────────────────────────────┘
```

This specification incorporates all findings from the Gemini audit and is ready for implementation.