Technical Interview Questions & Scaling Solutions

Potential Interview Questions

Code Structure & Design Questions

1. **"Why did you choose a class-based approach instead of just functions?"**
   - **Answer**: Encapsulation of state (cache, configuration), reusability, easier testing, and following OOP principles. The class maintains configuration and cache state across multiple operations.
2. **"What's the difference between `__init__` parameters and hard-coded values?"**
   - **Answer**: Configurability for different environments (dev/staging/prod), testability with mock values, and flexibility without code changes.
3. **"Why use a private method `_fetch_lei_data`?"**
   - **Answer**: Internal implementation detail that shouldn't be called directly by users. It maintains the class's interface contract and allows internal changes without breaking external code.
4. **"Explain the `-> Dict[str, Any]` syntax"**
   - **Answer**: Type hints that specify return type. `Dict[str, Any]` means dictionary with string keys and values of any type. Helps with IDE autocomplete, static analysis, and documentation.

Error Handling Questions

5. **"Why create a custom exception `LEIEnrichmentError`?"**
   - **Answer**: Specific error handling, clearer debugging, allows catching specific errors vs generic ones, better logging and monitoring.
6. **"What happens if the API returns malformed JSON?"**
   - **Answer**: `json.JSONDecodeError` is caught, logged as error, returns empty result, and caches it to avoid repeated failures.
7. **"Why continue processing if one LEI fails?"**
   - **Answer**: Resilience - partial success is better than total failure. Business requirement to process as much data as possible.

Performance & Optimization Questions

8. **"Why get unique LEIs instead of processing each row?"**
   - **Answer**: API call optimization. 1000 rows with 50 unique LEIs = 50 calls instead of 1000. Significant time and rate limit savings.
9. **"Explain the caching strategy. What are its limitations?"**
   - **Answer**: In-memory dictionary for session persistence, file-based for cross-session. Limitations: memory usage grows, no cache expiration, single-machine only.
10. **"What's exponential backoff and why use it?"**
    - **Answer**: `2^attempt` creates increasing delays (1s, 2s, 4s...). Prevents overwhelming struggling servers, gives them time to recover.

Data Processing Questions

11. **"Why use `df.copy()` instead of modifying the original?"**
    - **Answer**: Immutability principle, prevents side effects, allows original data to be used elsewhere, safer for debugging.
12. **"Explain the `axis=1` parameter in `apply()`"**
    - **Answer**: `axis=0` applies function to columns, `axis=1` applies to rows. We need row-wise calculation for transaction costs.
13. **"Why use `map()` instead of `apply()` for LEI lookups?"**
    - **Answer**: `map()` is faster for simple lookups, `apply()` is for complex functions. `map()` is optimized for dictionary/function mapping.

Business Logic Questions

14. **"Walk me through the transaction cost calculation"**
    - **Answer**: Country-specific formulas: GB uses interest calculation, NL uses inverse rate with absolute value, others default to 0.
15. **"How would you handle new country requirements?"**
    - **Answer**: Add new elif conditions, externalize rules to config file/database, or use strategy pattern for complex rules.

API Integration Questions

16. **"Why use `raise_for_status()`?"**
    - **Answer**: Automatically converts HTTP error codes (4xx, 5xx) to Python exceptions for consistent error handling.
17. **"How do you handle rate limits?"**
    - **Answer**: Sleep delay between calls, exponential backoff on failures, respect API documentation limits.
18. **"What if the API changes its response format?"**
    - **Answer**: Defensive programming with `.get()` methods, graceful degradation, version the API calls, comprehensive logging.

Scalability & Architecture Questions

19. **"How would you test this code?"**
    - **Answer**: Unit tests with mocked API calls, integration tests with test data, performance tests with large datasets.
20. **"How would you monitor this in production?"**
    - **Answer**: Structured logging, metrics (success rate, processing time), alerts on failures, cache hit rates.
21. **"What would you do if processing 1 million records?"**
    - **Answer**: Batch processing, async I/O, database caching, distributed processing, progress tracking.
22. **"How would you handle API key management?"**
    - **Answer**: Environment variables, secret management systems (AWS Secrets Manager, HashiCorp Vault), never hardcode in code.

Scaling Solutions Implementation

1. Async Processing with `asyncio` and `aiohttp`

python

```python
import asyncio
import aiohttp
import pandas as pd
from typing import List, Dict, Any
import time

class AsyncLEIEnricher:
    def __init__(self,
                 base_url: str = "https://api.gleif.org/api/v1/lei-records",
                 max_concurrent: int = 10,
                 rate_limit_delay: float = 0.1):
        self.base_url = base_url
        self.max_concurrent = max_concurrent
        self.rate_limit_delay = rate_limit_delay
        self.semaphore = asyncio.Semaphore(max_concurrent)
        self._lei_cache = {}

    async def _fetch_lei_data_async(self, session: aiohttp.ClientSession, lei_code: str) -> Dict[str, Any]:
        """Async version of LEI data fetching"""
        if lei_code in self._lei_cache:
            return self._lei_cache[lei_code]

        async with self.semaphore:  # Limit concurrent requests
            url = f"{self.base_url}?filter[lei]={lei_code}"

            try:
                async with session.get(url, timeout=aiohttp.ClientTimeout(total=30)) as response:
                    response.raise_for_status()
                    data = await response.json()

                    # Same data extraction logic as before
                    if 'data' in data and len(data['data']) > 0:
                        lei_record = data['data'][0]
                        attributes = lei_record.get('attributes', {})

                        legal_name = ''
                        entity = attributes.get('entity', {})
                        if entity and 'legalName' in entity:
                            legal_name = entity['legalName'].get('name', '')

                        result = {
                            'legalName': legal_name,
                            'bic': attributes.get('bic', [''])[0] if attributes.get('bic') else '',
                            'country': entity.get('legalAddress', {}).get('country', '') if entity else ''
                        }
```

```python
                }
            else:
                result = {'legalName': '', 'bic': '', 'country': ''}

            self._lei_cache[lei_code] = result
            await asyncio.sleep(self.rate_limit_delay)  # Async sleep
            return result

        except Exception as e:
            print(f"Error fetching {lei_code}: {e}")
            result = {'legalName': '', 'bic': '', 'country': ''}
            self._lei_cache[lei_code] = result
            return result

    async def enrich_dataset_async(self, input_data: pd.DataFrame) -> pd.DataFrame:
        """Async enrichment of entire dataset"""
        unique_leis = input_data['lei'].unique()

        async with aiohttp.ClientSession() as session:
            # Create tasks for all LEIs
            tasks = [
                self._fetch_lei_data_async(session, lei_code)
                for lei_code in unique_leis
            ]

            # Execute all tasks concurrently
            results = await asyncio.gather(*tasks, return_exceptions=True)

            # Build lei_info dictionary
            lei_info = {}
            for lei_code, result in zip(unique_leis, results):
                if isinstance(result, Exception):
                    lei_info[lei_code] = {'legalName': '', 'bic': '', 'country': ''}
                else:
                    lei_info[lei_code] = result

        # Apply results to DataFrame (same as before)
        enriched_data = input_data.copy()
        enriched_data['legalName'] = enriched_data['lei'].map(
            lambda x: lei_info.get(x, {}).get('legalName', '')
        )
        enriched_data['bic'] = enriched_data['lei'].map(
            lambda x: lei_info.get(x, {}).get('bic', '')
        )

        return enriched_data
```

```python
# Usage:
async def main_async():
    enricher = AsyncLEIEnricher(max_concurrent=10)
    df = pd.read_csv("sample_input.csv")

    start_time = time.time()
    enriched_df = await enricher.enrich_dataset_async(df)
    end_time = time.time()

    print(f"Async processing took: {end_time - start_time:.2f} seconds")
    return enriched_df

# Run async code:
# enriched_df = asyncio.run(main_async())
```

Key Async Concepts:

- `asyncio.Semaphore(10)`: Limits concurrent connections to prevent overwhelming the API
- `aiohttp.ClientSession`: Reuses connections for efficiency
- `asyncio.gather()`: Runs all tasks concurrently and waits for completion
- `async with`: Ensures proper resource cleanup

2. Database Caching with Redis

python

```python
import redis
import json
import pickle
from datetime import datetime, timedelta

class DatabaseCachedLEIEnricher:
    def __init__(self,
                 base_url: str = "https://api.gleif.org/api/v1/lei-records",
                 redis_host: str = "localhost",
                 redis_port: int = 6379,
                 cache_ttl: int = 86400):  # 24 hours
        self.base_url = base_url
        self.cache_ttl = cache_ttl

        # Redis connection with connection pooling
        self.redis_client = redis.ConnectionPool(
            host=redis_host,
            port=redis_port,
            decode_responses=True,
            max_connections=10
        )
        self.redis = redis.Redis(connection_pool=self.redis_client)

    def _get_cache_key(self, lei_code: str) -> str:
        """Generate Redis cache key"""
        return f"lei:v1:{lei_code}"

    def _fetch_from_cache(self, lei_code: str) -> Dict[str, Any]:
        """Fetch LEI data from Redis cache"""
        cache_key = self._get_cache_key(lei_code)
        cached_data = self.redis.get(cache_key)

        if cached_data:
            try:
                return json.loads(cached_data)
            except json.JSONDecodeError:
                # Handle corrupted cache data
                self.redis.delete(cache_key)
                return None
        return None

    def _store_in_cache(self, lei_code: str, data: Dict[str, Any]):
        """Store LEI data in Redis cache with TTL"""
        cache_key = self._get_cache_key(lei_code)
```

```python
        cache_data = {
            'data': data,
            'cached_at': datetime.utcnow().isoformat(),
            'version': '1.0'
        }

        try:
            self.redis.setex(
                cache_key,
                self.cache_ttl,
                json.dumps(cache_data)
            )
        except Exception as e:
            print(f"Failed to cache data for {lei_code}: {e}")

    def _fetch_lei_data(self, lei_code: str) -> Dict[str, Any]:
        """Enhanced fetch with database caching"""
        # Try cache first
        cached_result = self._fetch_from_cache(lei_code)
        if cached_result:
            return cached_result['data']

        # Fallback to API (same logic as original)
        result = self._fetch_from_api(lei_code)

        # Cache the result
        self._store_in_cache(lei_code, result)

        return result

    def get_cache_stats(self) -> Dict[str, int]:
        """Get cache performance statistics"""
        cache_keys = self.redis.keys("lei:v1:*")
        total_keys = len(cache_keys)

        # Get cache size in bytes
        cache_memory = sum(
            self.redis.memory_usage(key) or 0
            for key in cache_keys
        )

        return {
            'total_cached_leis': total_keys,
            'cache_memory_bytes': cache_memory,
            'cache_memory_mb': round(cache_memory / (1024 * 1024), 2)
        }
```

```
    def clear_cache(self, pattern: str = "lei:v1:*"):
        """Clear cache by pattern"""
        keys = self.redis.keys(pattern)
        if keys:
            self.redis.delete(*keys)
        return len(keys)
```

Database Caching Benefits:

- **Persistence**: Cache survives application restarts
- **Sharing**: Multiple application instances share cache
- **TTL**: Automatic expiration of stale data
- **Memory Management**: Redis handles memory optimization
- **Scalability**: Can handle millions of cache entries

3. Distributed Processing with Celery

python

```python
from celery import Celery, group
import pandas as pd
from typing import List
import numpy as np

# Celery configuration
celery_app = Celery(
    'lei_enrichment',
    broker='redis://localhost:6379/0',
    backend='redis://localhost:6379/0'
)

@celery_app.task(bind=True, max_retries=3)
def process_lei_batch(self, lei_codes: List[str]) -> Dict[str, Dict[str, Any]]:
    """Celery task to process a batch of LEI codes"""
    try:
        enricher = LEIDataEnricher()  # Create instance in worker
        results = {}

        for lei_code in lei_codes:
            try:
                results[lei_code] = enricher._fetch_lei_data(lei_code)
            except Exception as e:
                # Log error but continue processing other LEIs
                print(f"Failed to process {lei_code}: {e}")
                results[lei_code] = {'legalName': '', 'bic': '', 'country': ''}

        return results

    except Exception as exc:
        # Retry logic for entire batch
        print(f"Task failed, retrying: {exc}")
        raise self.retry(countdown=60 * (self.request.retries + 1))

class DistributedLEIEnricher:
    def __init__(self, batch_size: int = 50):
        self.batch_size = batch_size

    def create_batches(self, lei_codes: List[str]) -> List[List[str]]:
        """Split LEI codes into processing batches"""
        return [
            lei_codes[i:i + self.batch_size]
            for i in range(0, len(lei_codes), self.batch_size)
        ]
```

```python
    def enrich_dataset_distributed(self, input_data: pd.DataFrame) -> pd.DataFrame:
        """Distribute LEI enrichment across multiple workers"""
        unique_leis = input_data['lei'].unique().tolist()
        batches = self.create_batches(unique_leis)

        print(f"Processing {len(unique_leis)} LEIs in {len(batches)} batches")

        # Create Celery group for parallel execution
        job = group(process_lei_batch.s(batch) for batch in batches)

        # Execute all batches in parallel
        result = job.apply_async()

        # Wait for all batches to complete
        batch_results = result.get(timeout=300)  # 5 minute timeout

        # Combine results from all batches
        lei_info = {}
        for batch_result in batch_results:
            lei_info.update(batch_result)

        # Apply to DataFrame (same as before)
        enriched_data = input_data.copy()
        enriched_data['legalName'] = enriched_data['lei'].map(
            lambda x: lei_info.get(x, {}).get('legalName', '')
        )
        enriched_data['bic'] = enriched_data['lei'].map(
            lambda x: lei_info.get(x, {}).get('bic', '')
        )

        return enriched_data

# Usage:
def run_distributed_processing():
    enricher = DistributedLEIEnricher(batch_size=100)
    df = pd.read_csv("large_input.csv")

    enriched_df = enricher.enrich_dataset_distributed(df)
    enriched_df.to_csv("distributed_output.csv", index=False)

# Start Celery worker:
# celery -A lei_enricher worker --loglevel=info --concurrency=4
```

Distributed Processing Benefits:

- **Horizontal Scaling**: Add more worker machines
- **Fault Tolerance**: Failed tasks can be retried
- **Load Distribution**: Work spreads across available resources
- **Monitoring**: Built-in task monitoring and statistics

4. Complete Production Architecture

python

```python
import asyncio
import aioredis
from celery import Celery
import structlog
from prometheus_client import Counter, Histogram, start_http_server

class ProductionLEIEnricher:
    def __init__(self):
        # Structured logging
        self.logger = structlog.get_logger()

        # Metrics
        self.api_calls = Counter('lei_api_calls_total', 'Total API calls', ['status'])
        self.processing_time = Histogram('lei_processing_seconds', 'Processing time')

        # Start metrics server
        start_http_server(8000)

    async def enrich_with_monitoring(self, input_data: pd.DataFrame) -> pd.DataFrame:
        """Production enrichment with full monitoring"""
        with self.processing_time.time():
            try:
                # Log structured data
                self.logger.info(
                    "starting_enrichment",
                    record_count=len(input_data),
                    unique_leis=len(input_data['lei'].unique())
                )

                # Process data
                result = await self._process_with_circuit_breaker(input_data)

                # Success metrics
                self.api_calls.labels(status='success').inc()

                self.logger.info(
                    "enrichment_completed",
                    processed_records=len(result),
                    success_rate=self._calculate_success_rate(result)
                )

                return result

            except Exception as e:
```

```
            self.api_calls.labels(status='error').inc()
            self.logger.error("enrichment_failed", error=str(e))
            raise

    def _calculate_success_rate(self, df: pd.DataFrame) -> float:
        """Calculate enrichment success rate"""
        total = len(df)
        enriched = len(df[df['legalName'] != ''])
        return round((enriched / total) * 100, 2) if total > 0 else 0
```

Performance Comparison

Method	1000 LEIs	10000 LEIs	Memory Usage	Complexity
Synchronous	~100 seconds	~1000 seconds	Low	Simple
Async (10 concurrent)	~15 seconds	~150 seconds	Medium	Medium
Database Cached	~5 seconds	~50 seconds	Low	Medium
Distributed (4 workers)	~25 seconds	~250 seconds	High	Complex
Combined Approach	~3 seconds	~30 seconds	Medium	High

When to Use Each Approach:

- **Async**: High I/O operations, moderate scale (< 100k records)
- **Database Caching**: Repeated processing, multiple application instances
- **Distributed**: Very large datasets (> 1M records), horizontal scaling needs
- **Combined**: Production systems requiring high performance and reliability