Invoice Processing Pipeline Code Review

Critical Issues That Need Immediate Attention

1. Broken Exception Handling

The exception handling structure is fundamentally broken in multiple places:

```python
# Lines 194-225 - This except block is orphaned
except AuthenticationError as e:
    # This except has no corresponding try block above it
```

Fix: Each `except` block needs a proper `try` block. The code structure suggests you meant:

```python
for minibatch in tqdm(minibatches):
    try:
        quantity_removed_invoices = quantity_removed_invoices + (quantity_removal_template | llm.with_structured
        auth_attempts = 0
    except AuthenticationError:
        # handle auth error
    except Exception as e:
        # handle other errors
```

2. Logic Errors in Vector Database Initialization

Lines 85-96 have backwards logic:

```python
if not os.path.exists(f"{workingdir}/vector databases/firmwide policy title VDB"):
    category_idxr = FAISS.load_local(...)  # This will fail if path doesn't exist!
```

Fix: Should be:

```python
if os.path.exists(f"{workingdir}/vector databases/firmwide policy title VDB"):
    category_idxr = FAISS.load_local(...)
else:
    # Create new vector store
```

3. Undefined Variables

Multiple variables are used before being defined:

- `embeddings` (line 88) used before `generate_vector_DBs()` is called
- `global_p_dict` (line 92) used before being defined (line 139)
- `chunk_p_dict` (line 119) used before being defined (line 143)
- `time` imported but used as `time()` without `from time import time`

Code Structure & Organization Issues

4. Duplicate and Inconsistent Code

- The `generate_vector_DBs()` function is defined twice (lines 74 and 113)
- Similar patterns repeated for each processing phase without abstraction
- Inconsistent variable naming (`runtimelog` vs `runtimeLog` vs `errorlog` vs `errorLog`)

5. Poor Error Recovery

The retry logic is problematic:

```python
if (tmp_file_read_errors >= max_retries) and (len(batch) == 1):
    # Just skips the file - no proper logging or handling
```

Better approach:

```python
class InvoiceProcessingError(Exception):
    pass

def process_with_retry(func, data, max_retries=3):
    for attempt in range(max_retries):
        try:
            return func(data)
        except AuthenticationError:
            if attempt == max_retries - 1:
                raise
            refresh_auth_token()
        except Exception as e:
            logger.error(f"Attempt {attempt + 1} failed: {e}")
            if attempt == max_retries - 1:
                raise InvoiceProcessingError(f"Failed after {max_retries} attempts: {e}")
```

Performance & Efficiency Issues

6. Inefficient DataFrame Operations

Lines 235-238 create a DataFrame with complex dictionary comprehensions that are hard to read and debug:

```python
pd.DataFrame({
    'file_name': invoice_idx[:i],  # This slice is wrong - should be invoice_idx
    'dict(invoice_scrapes[i])' for i in range(len(invoice_idx))  # Syntax error
})
```

Fix:

```python
def create_phase_dataframe(invoice_idx, invoice_scrapes, phase_name):
    data = []
    for i, (filename, scrape) in enumerate(zip(invoice_idx, invoice_scrapes)):
        row = {'file_name': filename, 'invoice_id': i}
        row.update(scrape.__dict__ if hasattr(scrape, '__dict__') else scrape)
        data.append(row)

    df = pd.DataFrame(data)
    df.to_excel(f'{persistdir}/phase_{phase_name}.xlsx', index=False)
    return df
```

7. Memory Inefficiency

- Lists like `invoice_scrapes` grow indefinitely without cleanup
- Vector databases are regenerated unnecessarily
- Large embeddings are computed multiple times

Code Quality & Maintainability

8. Magic Numbers and Hard-coded Values

```python
step_size = 100  # Why 100? What happens if we change this?
max_retries = 3  # Should be configurable
```

Improve with configuration:

```python
```

```python
@dataclass
class ProcessingConfig:
    step_size: int = 100
    max_retries: int = 3
    embedding_model: str = "text-embedding-ada-002"
    llm_deployment: str = "gpt-4o-2024-08-06"

    @classmethod
    def from_env(cls, env_path: str) -> 'ProcessingConfig':
        # Load from environment file
        pass
```

9. Poor Logging Strategy

Multiple log files with inconsistent formatting:

```python
runtimelog.write(f"Time to establish connection to OpenAI: {delta_t}\n")
errorLog.write("Authentication failed. Generating new access token.\n")
```

Better logging:

```python
import logging

def setup_logging(persistdir):
    logging.basicConfig(
        level=logging.INFO,
        format='%(asctime)s - %(name)s - %(levelname)s - %(message)s',
        handlers=[
            logging.FileHandler(f'{persistdir}/pipeline.log'),
            logging.StreamHandler()
        ]
    )
    return logging.getLogger(__name__)

logger = setup_logging(persistdir)
logger.info(f"Connection established in {delta_t:.2f}s")
```

Suggested Refactoring

10. Create a Proper Class Structure

```python
```

```python
class InvoiceProcessingPipeline:
    def __init__(self, config: ProcessingConfig):
        self.config = config
        self.logger = setup_logging(config.persist_dir)
        self.llm = self._setup_llm()
        self.vector_stores = self._setup_vector_stores()

    def process_invoices(self, invoice_paths: List[str]) -> pd.DataFrame:
        """Main processing pipeline"""
        results = []

        # Phase 1: OCR
        scraped_data = self._ocr_phase(invoice_paths)

        # Phase 2: Table conversion
        extracted_data = self._extraction_phase(scraped_data)

        # Phase 3: Translation
        translated_data = self._translation_phase(extracted_data)

        # Phase 4: Cleaning
        cleaned_data = self._cleaning_phase(translated_data)

        # Phase 5: RAG decision
        final_results = self._rag_decision_phase(cleaned_data)

        return self._create_output_dataframe(final_results)

    def _ocr_phase(self, invoice_paths: List[str]) -> List[dict]:
        """OCR processing with proper error handling"""
        # Implementation here
        pass
```

11. Better Error Handling with Context Managers

```python
```

```python
from contextlib import contextmanager

@contextmanager
def auth_retry_context(max_retries=3):
    """Context manager for handling authentication retries"""
    retries = 0
    while retries < max_retries:
        try:
            yield
            break
        except AuthenticationError:
            retries += 1
            if retries >= max_retries:
                raise
            refresh_auth_token()
            logger.warning(f"Auth retry {retries}/{max_retries}")

# Usage:
with auth_retry_context():
    results = llm.batch(batch_data)
```

12. Input Validation

```python
def validate_inputs(invoice_paths: List[str], metadata_df: pd.DataFrame) -> None:
    """Validate all inputs before processing"""
    if not invoice_paths:
        raise ValueError("No invoice paths provided")

    missing_files = [p for p in invoice_paths if not os.path.exists(p)]
    if missing_files:
        raise FileNotFoundError(f"Missing files: {missing_files}")

    if metadata_df.empty:
        raise ValueError("Metadata DataFrame is empty")
```

Security & Best Practices

13. Credential Management

The current approach exposes credentials in code. Use proper secret management:

```python
```

```python
from azure.keyvault.secrets import SecretClient
from azure.identity import DefaultAzureCredential

def get_secure_credentials():
    credential = DefaultAzureCredential()
    secret_client = SecretClient(vault_url="https://your-vault.vault.azure.net/", credential=credential)

    return {
        'api_key': secret_client.get_secret("openai-api-key").value,
        'endpoint': secret_client.get_secret("openai-endpoint").value
    }
```

14. Resource Cleanup

Add proper cleanup for file handles and resources:

```python
class ManagedPipeline:
    def __enter__(self):
        self._setup_resources()
        return self

    def __exit__(self, exc_type, exc_val, exc_tb):
        self._cleanup_resources()

    def _cleanup_resources(self):
        for log_file in [self.runtime_log, self.error_log, self.monitor_log]:
            if log_file and not log_file.closed:
                log_file.close()
```

Testing & Monitoring

15. Add Progress Tracking

```python
```

```python
from tqdm import tqdm
import time

class ProgressTracker:
    def __init__(self, total_phases=5):
        self.phase_times = {}
        self.current_phase = 0
        self.total_phases = total_phases

    def start_phase(self, phase_name: str):
        self.current_phase += 1
        self.phase_start = time.time()
        print(f"Starting Phase {self.current_phase}/{self.total_phases}: {phase_name}")

    def end_phase(self, phase_name: str):
        duration = time.time() - self.phase_start
        self.phase_times[phase_name] = duration
        print(f"Completed {phase_name} in {duration:.2f}s")
```

Summary of Priority Fixes

1. **Fix the broken exception handling** - This will crash the code
2. **Fix the vector database logic** - Currently creates/loads incorrectly
3. **Define missing variables** - Import `time`, define `embeddings` before use
4. **Restructure into proper classes** - Makes testing and maintenance easier
5. **Add proper logging** - Use Python's logging module
6. **Implement retry logic properly** - Current retry mechanism is fragile
7. **Add input validation** - Fail fast with clear error messages
8. **Use configuration objects** - Make the system configurable
9. **Add progress tracking** - Essential for long-running processes
10. **Implement proper cleanup** - Prevent resource leaks

The code shows good understanding of the LLM pipeline concepts, but needs significant refactoring for production use. Focus on the critical issues first, then gradually improve the architecture.