Data Engineering Best Practices & Implementation Guide

Table of Contents

1. Code Organization & Structure
2. Error Handling Strategies
3. Performance Optimization
4. Testing Strategies
5. Monitoring & Logging
6. Security Implementation
7. Data Quality Management
8. Deployment & Operations

Code Organization & Structure

1. Project Structure

```
facilities_import/
├── src/
│   ├── etl_pipeline/
│   │   ├── __init__.py
│   │   ├── config.py              # Configuration management
│   │   ├── pipeline.py            # Main ETL orchestrator
│   │   ├── extractors/            # Data extraction modules
│   │   │   ├── __init__.py
│   │   │   ├── json_extractor.py
│   │   │   └── api_extractor.py
│   │   ├── transformers/          # Data transformation modules
│   │   │   ├── __init__.py
│   │   │   ├── facility_transformer.py
│   │   │   └── geographic_transformer.py
│   │   ├── loaders/               # Data loading modules
│   │   │   ├── __init__.py
│   │   │   └── database_loader.py
│   │   └── utils/                 # Utility functions
│   │       ├── __init__.py
│   │       ├── validators.py
│   │       ├── encoders.py
│   │       └── helpers.py
│   ├── data/
│   │   ├── raw/                   # Raw data files
│   │   ├── processed/             # Processed data files
│   │   └── validated/             # Validated data files
│   └── scripts/
│       ├── production_importer.py
│       └── deploy_etl.py
```

```
├── config/
│   ├── etl_requirements.txt
│   ├── production.yaml
│   └── development.yaml
├── tests/
│   ├── unit/
│   ├── integration/
│   └── e2e/
├── logs/
│   ├── etl/
│   ├── errors/
│   └── processing/
└── docs/
    ├── api/
    ├── architecture/
    └── user_guides/
```

2. Configuration Management

```python
# config.py
from dataclasses import dataclass
from typing import Dict, Any, Optional
import yaml
import os

@dataclass
class DatabaseConfig:
    host: str
    port: int
    database: str
    username: str
    password: str
    pool_size: int = 10
    timeout: int = 30

@dataclass
class ETLConfig:
    batch_size: int = 50
    max_retries: int = 3
    timeout_seconds: int = 300
    enable_parallel_processing: bool = True
    max_concurrent_tasks: int = 4

class ConfigManager:
    def __init__(self, config_path: str = None):
        self.config_path = config_path or os.getenv('CONFIG_PATH',
'config/production.yaml')
        self._config = self._load_config()

    def _load_config(self) -> Dict[str, Any]:
        """Load configuration from YAML file"""
```

```
        with open(self.config_path, 'r') as file:
            return yaml.safe_load(file)

    def get_database_config(self) -> DatabaseConfig:
        """Get database configuration"""
        db_config = self._config['database']
        return DatabaseConfig(**db_config)

    def get_etl_config(self) -> ETLConfig:
        """Get ETL configuration"""
        etl_config = self._config['etl']
        return ETLConfig(**etl_config)
```

Error Handling Strategies

1. Custom Exception Hierarchy

```
# exceptions.py
class ETLException(Exception):
    """Base exception for ETL operations"""
    def __init__(self, message: str, error_code: str = None, details: Dict
= None):
        super().__init__(message)
        self.message = message
        self.error_code = error_code
        self.details = details or {}

class DataValidationError(ETLException):
    """Data validation failed"""
    pass

class DataTransformationError(ETLException):
    """Data transformation failed"""
    pass

class DatabaseConnectionError(ETLException):
    """Database connection failed"""
    pass

class DataQualityError(ETLException):
    """Data quality check failed"""
    pass
```

2. Circuit Breaker Pattern

```
# circuit_breaker.py
import time
from enum import Enum
```

```python
from typing import Callable, Any

class CircuitState(Enum):
    CLOSED = "CLOSED"
    OPEN = "OPEN"
    HALF_OPEN = "HALF_OPEN"

class CircuitBreaker:
    def __init__(self, failure_threshold: int = 5, timeout: int = 60,
                 expected_exception: type = Exception):
        self.failure_threshold = failure_threshold
        self.timeout = timeout
        self.expected_exception = expected_exception
        self.failure_count = 0
        self.last_failure_time = None
        self.state = CircuitState.CLOSED

    def call(self, func: Callable, *args, **kwargs) -> Any:
        """Execute function with circuit breaker protection"""
        if self.state == CircuitState.OPEN:
            if self._should_attempt_reset():
                self.state = CircuitState.HALF_OPEN
            else:
                raise CircuitBreakerError("Circuit breaker is OPEN")

        try:
            result = func(*args, **kwargs)
            self._on_success()
            return result
        except self.expected_exception as e:
            self._on_failure()
            raise e

    def _should_attempt_reset(self) -> bool:
        """Check if enough time has passed to attempt reset"""
        return (time.time() - self.last_failure_time) > self.timeout

    def _on_success(self):
        """Handle successful execution"""
        self.failure_count = 0
        self.state = CircuitState.CLOSED

    def _on_failure(self):
        """Handle failed execution"""
        self.failure_count += 1
        self.last_failure_time = time.time()

        if self.failure_count >= self.failure_threshold:
            self.state = CircuitState.OPEN
```

3. Retry Mechanism

```python
# retry.py
import time
import random
from typing import Callable, Any, Optional
from functools import wraps

def retry_with_exponential_backoff(
    max_retries: int = 3,
    base_delay: float = 1.0,
    max_delay: float = 60.0,
    exponential_base: float = 2.0,
    jitter: bool = True,
    exceptions: tuple = (Exception,)
):
    """Decorator for retrying functions with exponential backoff"""
    def decorator(func: Callable) -> Callable:
        @wraps(func)
        def wrapper(*args, **kwargs) -> Any:
            last_exception = None

            for attempt in range(max_retries + 1):
                try:
                    return func(*args, **kwargs)
                except exceptions as e:
                    last_exception = e

                    if attempt == max_retries:
                        raise e

                    # Calculate delay with exponential backoff
                    delay = min(base_delay * (exponential_base ** attempt),
max_delay)

                    # Add jitter to prevent thundering herd
                    if jitter:
                        delay *= (0.5 + random.random() * 0.5)

                    time.sleep(delay)

            raise last_exception
        return wrapper
    return decorator

# Usage example
@retry_with_exponential_backoff(max_retries=3, exceptions=
(DatabaseConnectionError,))
def database_operation():
    # Database operation that might fail
    pass
```

Performance Optimization

1. Database Optimization

```python
# database_optimizer.py
from django.db import connection
from contextlib import contextmanager
import psycopg2

class DatabaseOptimizer:
    def __init__(self):
        self.connection_pool = self._create_connection_pool()

    def _create_connection_pool(self):
        """Create database connection pool"""
        return psycopg2.pool.SimpleConnectionPool(
            minconn=1,
            maxconn=20,
            host='localhost',
            database='gvrc_admin_production',
            user='gvrc_user',
            password='gvrc_password123'
        )

    @contextmanager
    def get_connection(self):
        """Get connection from pool"""
        conn = self.connection_pool.getconn()
        try:
            yield conn
        finally:
            self.connection_pool.putconn(conn)

    def bulk_insert_facilities(self, facilities: list):
        """Optimized bulk insert for facilities"""
        with self.get_connection() as conn:
            with conn.cursor() as cursor:
                # Use COPY for maximum performance
                cursor.execute("""
                    CREATE TEMP TABLE temp_facilities (
                        facility_code VARCHAR(50),
                        registration_number VARCHAR(100),
                        facility_name VARCHAR(255),
                        facility_type VARCHAR(100),
                        operational_status_id INTEGER,
                        ward_id INTEGER,
                        created_by INTEGER,
                        updated_by INTEGER,
                        is_active BOOLEAN
                    )
                """)

                # Prepare data for COPY
                data = []
```

```
                for facility in facilities:
                    data.append((
                        facility['facility_code'],
                        facility['registration_number'],
                        facility['facility_name'],
                        facility.get('facility_type'),
                        facility.get('operational_status_id'),
                        facility.get('ward_id'),
                        facility.get('created_by'),
                        facility.get('updated_by'),
                        facility.get('is_active', True)
                    ))

                # Use COPY for bulk insert
                cursor.executemany(
                    "INSERT INTO temp_facilities VALUES (%s, %s, %s, %s,
%s, %s, %s, %s, %s)",
                    data
                )

                # Insert from temp table with conflict resolution
                cursor.execute("""
                    INSERT INTO facilities (
                        facility_code, registration_number, facility_name,
                        facility_type, operational_status_id, ward_id,
                        created_by, updated_by, is_active
                    )
                    SELECT * FROM temp_facilities
                    ON CONFLICT (facility_code) DO UPDATE SET
                        registration_number = EXCLUDED.registration_number,
                        facility_name = EXCLUDED.facility_name,
                        updated_at = CURRENT_TIMESTAMP
                """)

                conn.commit()
```

2. Memory Management

```
# memory_manager.py
import psutil
import gc
from typing import Generator, Any

class MemoryManager:
    def __init__(self, max_memory_mb: int = 512):
        self.max_memory_mb = max_memory_mb
        self.process = psutil.Process()

    def get_memory_usage(self) -> float:
        """Get current memory usage in MB"""
        return self.process.memory_info().rss / 1024 / 1024
```

```
    def is_memory_available(self) -> bool:
        """Check if memory usage is within limits"""
        return self.get_memory_usage() < self.max_memory_mb

    def force_garbage_collection(self):
        """Force garbage collection to free memory"""
        gc.collect()

    def process_in_chunks(self, data: list, chunk_size: int = 1000) ->
Generator[list, None, None]:
        """Process data in memory-efficient chunks"""
        for i in range(0, len(data), chunk_size):
            chunk = data[i:i + chunk_size]

            # Check memory before processing chunk
            if not self.is_memory_available():
                self.force_garbage_collection()

                if not self.is_memory_available():
                    raise MemoryError(f"Memory usage exceeded
{self.max_memory_mb}MB")

            yield chunk
```

3. Caching Strategy

```
# cache_manager.py
from functools import lru_cache
import redis
import json
from typing import Any, Optional

class CacheManager:
    def __init__(self, redis_host: str = 'localhost', redis_port: int =
6379):
        self.redis_client = redis.Redis(host=redis_host, port=redis_port,
decode_responses=True)
        self.local_cache = {}

    def get(self, key: str) -> Optional[Any]:
        """Get value from cache (Redis -> Local)"""
        # Try local cache first
        if key in self.local_cache:
            return self.local_cache[key]

        # Try Redis cache
        try:
            value = self.redis_client.get(key)
            if value:
                parsed_value = json.loads(value)
```

```
                self.local_cache[key] = parsed_value
                return parsed_value
            except Exception:
                pass

        return None

    def set(self, key: str, value: Any, ttl: int = 3600):
        """Set value in cache (Local + Redis)"""
        # Set in local cache
        self.local_cache[key] = value

        # Set in Redis cache
        try:
            self.redis_client.setex(key, ttl, json.dumps(value))
        except Exception:
            pass

    @lru_cache(maxsize=128)
    def get_county_by_name(self, county_name: str):
        """Cached county lookup"""
        # This will be cached by lru_cache
        return County.objects.get(county_name=county_name)
```

Testing Strategies

1. Unit Testing

```
# tests/unit/test_extractors.py
import pytest
from unittest.mock import Mock, patch, mock_open
import json
from src.etl_pipeline.extractors.json_extractor import JSONExtractor

class TestJSONExtractor:
    def setup_method(self):
        self.extractor = JSONExtractor()
        self.sample_data = {
            'facilities': [
                {
                    'name': 'Test Hospital',
                    'code': 'TH-001',
                    'location': {
                        'county': 'Nairobi',
                        'constituency': 'Westlands'
                    }
                }
            ]
        }
```

```
    def test_extract_facilities_success(self):
        """Test successful facility extraction"""
        with patch('builtins.open',
mock_open(read_data=json.dumps(self.sample_data))):
            result = self.extractor.extract_facilities('test.json')
            assert len(result) == 1
            assert result[0]['name'] == 'Test Hospital'

    def test_extract_facilities_file_not_found(self):
        """Test handling of missing file"""
        with patch('builtins.open', side_effect=FileNotFoundError):
            result = self.extractor.extract_facilities('nonexistent.json')
            assert result == []

    def test_extract_facilities_invalid_json(self):
        """Test handling of invalid JSON"""
        with patch('builtins.open', mock_open(read_data='invalid json')):
            result = self.extractor.extract_facilities('invalid.json')
            assert result == []

    @pytest.mark.parametrize("input_data,expected_count", [
        ({'facilities': []}, 0),
        ({'facilities': [{'name': 'Test'}]}, 1),
        ({'data': [{'name': 'Test'}]}, 1),
        ([{'name': 'Test'}], 1),
    ])
    def test_extract_facilities_different_formats(self, input_data,
expected_count):
        """Test extraction with different JSON formats"""
        with patch('builtins.open',
mock_open(read_data=json.dumps(input_data))):
            result = self.extractor.extract_facilities('test.json')
            assert len(result) == expected_count
```

2. Integration Testing

```
# tests/integration/test_etl_pipeline.py
import pytest
from django.test import TestCase
from django.db import transaction
from src.etl_pipeline.pipeline import ETLPipeline
from facilities.models import Facility, County, Constituency, Ward

class TestETLPipeline(TestCase):
    def setUp(self):
        self.pipeline = ETLPipeline()
        self.sample_facilities = [
            {
                'facility_name': 'Test Hospital 1',
                'facility_code': 'TH-001',
                'location': {
```

```
                    'county': 'Nairobi',
                    'constituency': 'Westlands',
                    'ward': 'Parklands'
                }
            },
            {
                'facility_name': 'Test Hospital 2',
                'facility_code': 'TH-002',
                'location': {
                    'county': 'Nairobi',
                    'constituency': 'Westlands',
                    'ward': 'Parklands'
                }
            }
        ]

    def test_full_etl_pipeline(self):
        """Test complete ETL pipeline execution"""
        # Execute pipeline
        result = self.pipeline.run(self.sample_facilities)

        # Verify results
        assert result['success'] == True
        assert result['processed_count'] == 2
        assert result['error_count'] == 0

        # Verify data in database
        facilities = Facility.objects.all()
        assert facilities.count() == 2

        # Verify geographic mapping
        nairobi = County.objects.get(county_name='Nairobi')
        westlands = Constituency.objects.get(constituency_name='Westlands')
        parklands = Ward.objects.get(ward_name='Parklands')

        assert nairobi is not None
        assert westlands.county == nairobi
        assert parklands.constituency == westlands

    def test_etl_pipeline_with_errors(self):
        """Test ETL pipeline with invalid data"""
        invalid_facilities = [
            {
                'facility_name': '',  # Invalid: empty name
                'facility_code': 'TH-001',
                'location': {'county': 'Nairobi'}
            },
            {
                'facility_name': 'Test Hospital',
                'facility_code': '',  # Invalid: empty code
                'location': {'county': 'Nairobi'}
            }
        ]
```

```
        result = self.pipeline.run(invalid_facilities)

        # Should handle errors gracefully
        assert result['success'] == False
        assert result['error_count'] > 0
        assert len(result['errors']) > 0
```

3. End-to-End Testing

```
# tests/e2e/test_production_importer.py
import pytest
import subprocess
import os
from pathlib import Path

class TestProductionImporter:
    def test_production_importer_execution(self):
        """Test production importer script execution"""
        script_path = Path(__file__).parent.parent.parent / 'src' /
'scripts' / 'production_json_importer.py'

        # Run the production importer
        result = subprocess.run(
            ['python', str(script_path)],
            capture_output=True,
            text=True,
            cwd=os.getcwd()
        )

        # Verify execution
        assert result.returncode == 0
        assert 'ETL Pipeline completed successfully' in result.stdout

    def test_database_migration(self):
        """Test database migration process"""
        # Test migration commands
        migrate_result = subprocess.run(
            ['python', 'manage.py', 'migrate'],
            capture_output=True,
            text=True
        )

        assert migrate_result.returncode == 0
        assert 'No migrations to apply' in migrate_result.stdout or
'Applying' in migrate_result.stdout
```

Monitoring & Logging

1. Structured Logging

```python
# logging_config.py
import structlog
import logging
from pathlib import Path

def setup_logging(log_level: str = 'INFO', log_dir: str = 'logs'):
    """Configure structured logging"""
    log_dir = Path(log_dir)
    log_dir.mkdir(exist_ok=True)

    # Configure structlog
    structlog.configure(
        processors=[
            structlog.stdlib.filter_by_level,
            structlog.stdlib.add_logger_name,
            structlog.stdlib.add_log_level,
            structlog.stdlib.PositionalArgumentsFormatter(),
            structlog.processors.TimeStamper(fmt="iso"),
            structlog.processors.StackInfoRenderer(),
            structlog.processors.format_exc_info,
            structlog.processors.UnicodeDecoder(),
            structlog.processors.JSONRenderer()
        ],
        context_class=dict,
        logger_factory=structlog.stdlib.LoggerFactory(),
        wrapper_class=structlog.stdlib.BoundLogger,
        cache_logger_on_first_use=True,
    )

    # Configure file handlers
    file_handler = logging.FileHandler(log_dir / 'etl_pipeline.log')
    file_handler.setLevel(getattr(logging, log_level))

    error_handler = logging.FileHandler(log_dir / 'errors.log')
    error_handler.setLevel(logging.ERROR)

    # Configure root logger
    root_logger = logging.getLogger()
    root_logger.setLevel(getattr(logging, log_level))
    root_logger.addHandler(file_handler)
    root_logger.addHandler(error_handler)

# Usage
logger = structlog.get_logger()

def log_etl_event(event_type: str, **kwargs):
    """Log ETL events with context"""
    logger.info(
        event_type,
        pipeline_id=kwargs.get('pipeline_id'),
        facility_id=kwargs.get('facility_id'),
        processing_time=kwargs.get('processing_time'),
```

```
            status=kwargs.get('status'),
            **kwargs
        )
```

2. Metrics Collection

```
# metrics_collector.py
import time
from dataclasses import dataclass
from typing import Dict, Any
import psutil

@dataclass
class ProcessingMetrics:
    records_processed: int = 0
    processing_time: float = 0.0
    error_count: int = 0
    quality_score: float = 0.0
    memory_usage_mb: float = 0.0
    cpu_usage_percent: float = 0.0

class MetricsCollector:
    def __init__(self):
        self.metrics = ProcessingMetrics()
        self.start_time = None
        self.process = psutil.Process()

    def start_processing(self):
        """Start processing timer"""
        self.start_time = time.time()

    def end_processing(self):
        """End processing timer"""
        if self.start_time:
            self.metrics.processing_time = time.time() - self.start_time

    def record_batch_processing(self, batch_size: int, errors: list):
        """Record batch processing metrics"""
        self.metrics.records_processed += batch_size
        self.metrics.error_count += len(errors)

        # Update system metrics
        self.metrics.memory_usage_mb = self.process.memory_info().rss /
1024 / 1024
        self.metrics.cpu_usage_percent = self.process.cpu_percent()

        # Calculate quality score
        if self.metrics.records_processed > 0:
            error_rate = self.metrics.error_count /
self.metrics.records_processed
            self.metrics.quality_score = max(0, (1 - error_rate) * 100)
```

```
    def get_metrics_summary(self) -> Dict[str, Any]:
        """Get metrics summary"""
        return {
            'total_records_processed': self.metrics.records_processed,
            'total_processing_time': round(self.metrics.processing_time,
2),
            'average_processing_time_per_record': round(
                self.metrics.processing_time / max(1,
self.metrics.records_processed), 4
            ),
            'total_errors': self.metrics.error_count,
            'error_rate': round(
                self.metrics.error_count / max(1,
self.metrics.records_processed) * 100, 2
            ),
            'quality_score': round(self.metrics.quality_score, 2),
            'memory_usage_mb': round(self.metrics.memory_usage_mb, 2),
            'cpu_usage_percent': round(self.metrics.cpu_usage_percent, 2)
        }
```

Security Implementation

1. Data Encryption

```python
# encryption.py
from cryptography.fernet import Fernet
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.kdf.pbkdf2 import PBKDF2HMAC
import base64
import os

class DataEncryption:
    def __init__(self, password: str = None):
        self.password = password or os.getenv('ENCRYPTION_PASSWORD')
        self.key = self._derive_key()
        self.cipher = Fernet(self.key)

    def _derive_key(self) -> bytes:
        """Derive encryption key from password"""
        password = self.password.encode()
        salt = b'gvrc_salt_2025'  # In production, use random salt
        kdf = PBKDF2HMAC(
            algorithm=hashes.SHA256(),
            length=32,
            salt=salt,
            iterations=100000,
        )
        key = base64.urlsafe_b64encode(kdf.derive(password))
        return key
```

```
    def encrypt_sensitive_data(self, data: dict) -> dict:
        """Encrypt sensitive fields in data"""
        sensitive_fields = ['phone_number', 'email', 'address',
'contact_value']
        encrypted_data = data.copy()

        for field in sensitive_fields:
            if field in data and data[field]:
                try:
                    encrypted_data[field] = self.cipher.encrypt(
                        str(data[field]).encode()
                    ).decode()
                except Exception as e:
                    # Log encryption error but don't fail
                    print(f"Encryption error for field {field}: {e}")

        return encrypted_data

    def decrypt_sensitive_data(self, encrypted_data: dict) -> dict:
        """Decrypt sensitive fields in data"""
        sensitive_fields = ['phone_number', 'email', 'address',
'contact_value']
        decrypted_data = encrypted_data.copy()

        for field in sensitive_fields:
            if field in encrypted_data and encrypted_data[field]:
                try:
                    decrypted_data[field] = self.cipher.decrypt(
                        encrypted_data[field].encode()
                    ).decode()
                except Exception:
                    # Handle decryption errors gracefully
                    decrypted_data[field] = None

        return decrypted_data
```

2. Access Control

```
# access_control.py
from enum import Enum
from typing import List, Dict
from functools import wraps

class Permission(Enum):
    READ = "read"
    WRITE = "write"
    DELETE = "delete"
    MANAGE_USERS = "manage_users"
    VIEW_LOGS = "view_logs"
    MANAGE_CONFIG = "manage_config"
```

```python
class Role(Enum):
    DATA_ENGINEER = "data_engineer"
    DATA_ANALYST = "data_analyst"
    ADMIN = "admin"
    VIEWER = "viewer"

class AccessControl:
    def __init__(self):
        self.role_permissions = {
            Role.DATA_ENGINEER: [
                Permission.READ, Permission.WRITE, Permission.VIEW_LOGS
            ],
            Role.DATA_ANALYST: [Permission.READ],
            Role.ADMIN: [
                Permission.READ, Permission.WRITE, Permission.DELETE,
                Permission.MANAGE_USERS, Permission.VIEW_LOGS,
Permission.MANAGE_CONFIG
            ],
            Role.VIEWER: [Permission.READ]
        }

    def has_permission(self, user_role: Role, permission: Permission) ->
bool:
        """Check if user role has specific permission"""
        if user_role not in self.role_permissions:
            return False
        return permission in self.role_permissions[user_role]

    def require_permission(self, permission: Permission):
        """Decorator to require specific permission"""
        def decorator(func):
            @wraps(func)
            def wrapper(*args, **kwargs):
                # In real implementation, get user role from session/token
                user_role = kwargs.get('user_role', Role.VIEWER)

                if not self.has_permission(user_role, permission):
                    raise PermissionError(f"Permission {permission.value}
required")

                return func(*args, **kwargs)
            return wrapper
        return decorator

# Usage example
access_control = AccessControl()

@access_control.require_permission(Permission.WRITE)
def create_facility(facility_data: dict, user_role: Role = Role.VIEWER):
    """Create facility with write permission required"""
    # Implementation here
    pass
```

Data Quality Management

1. Data Validation Framework

```python
# data_validation.py
from typing import Dict, List, Any, Optional
import re
from dataclasses import dataclass

@dataclass
class ValidationResult:
    is_valid: bool
    errors: List[str]
    warnings: List[str]
    quality_score: float

class DataValidator:
    def __init__(self):
        self.validation_rules = {
            'facility_name': self.validate_facility_name,
            'facility_code': self.validate_facility_code,
            'phone_number': self.validate_phone_number,
            'email': self.validate_email,
            'coordinates': self.validate_coordinates
        }

    def validate_facility(self, facility_data: dict) -> ValidationResult:
        """Validate complete facility data"""
        errors = []
        warnings = []

        # Required field validation
        required_fields = ['facility_name', 'facility_code']
        for field in required_fields:
            if not facility_data.get(field):
                errors.append(f"Required field '{field}' is missing")

        # Field-specific validation
        for field, validator in self.validation_rules.items():
            if field in facility_data:
                field_result = validator(facility_data[field])
                if not field_result['is_valid']:
                    errors.extend(field_result['errors'])
                warnings.extend(field_result['warnings'])

        # Calculate quality score
        total_fields = len(facility_data)
        error_fields = len(errors)
        quality_score = max(0, (total_fields - error_fields) / total_fields
* 100)
```

```
        return ValidationResult(
            is_valid=len(errors) == 0,
            errors=errors,
            warnings=warnings,
            quality_score=quality_score
        )

    def validate_facility_name(self, name: str) -> Dict[str, Any]:
        """Validate facility name"""
        errors = []
        warnings = []

        if not name or not name.strip():
            errors.append("Facility name cannot be empty")
        elif len(name) < 3:
            errors.append("Facility name too short")
        elif len(name) > 255:
            errors.append("Facility name too long")
        elif not re.match(r'^[a-zA-Z0-9\s\-\.]+$', name):
            warnings.append("Facility name contains special characters")

        return {
            'is_valid': len(errors) == 0,
            'errors': errors,
            'warnings': warnings
        }

    def validate_facility_code(self, code: str) -> Dict[str, Any]:
        """Validate facility code format"""
        errors = []
        warnings = []

        if not code or not code.strip():
            errors.append("Facility code cannot be empty")
        elif not re.match(r'^[A-Z]{2,4}-\d{3,6}$', code):
            errors.append("Facility code must match pattern: XX-XXX or XXX-
XXXX")

        return {
            'is_valid': len(errors) == 0,
            'errors': errors,
            'warnings': warnings
        }

    def validate_phone_number(self, phone: str) -> Dict[str, Any]:
        """Validate phone number format"""
        errors = []
        warnings = []

        if not phone:
            return {'is_valid': True, 'errors': [], 'warnings': []}

        # Remove spaces and special characters
```

```
        clean_phone = re.sub(r'[^\d+]', '', phone)

        if not re.match(r'^\+254\d{9}$', clean_phone):
            if not re.match(r'^0\d{9}$', clean_phone):
                errors.append("Invalid phone number format")
            else:
                warnings.append("Phone number should include country code
(+254)")

        return {
            'is_valid': len(errors) == 0,
            'errors': errors,
            'warnings': warnings
        }

    def validate_email(self, email: str) -> Dict[str, Any]:
        """Validate email format"""
        errors = []
        warnings = []

        if not email:
            return {'is_valid': True, 'errors': [], 'warnings': []}

        email_pattern = r'^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$'
        if not re.match(email_pattern, email):
            errors.append("Invalid email format")

        return {
            'is_valid': len(errors) == 0,
            'errors': errors,
            'warnings': warnings
        }

    def validate_coordinates(self, coords: dict) -> Dict[str, Any]:
        """Validate geographic coordinates"""
        errors = []
        warnings = []

        if not coords:
            return {'is_valid': True, 'errors': [], 'warnings': []}

        lat = coords.get('latitude')
        lon = coords.get('longitude')

        if lat is None or lon is None:
            errors.append("Both latitude and longitude are required")
        else:
            if not (-90 <= lat <= 90):
                errors.append("Latitude must be between -90 and 90")
            if not (-180 <= lon <= 180):
                errors.append("Longitude must be between -180 and 180")

        return {
            'is_valid': len(errors) == 0,
```

```
            'errors': errors,
            'warnings': warnings
        }
```

Deployment & Operations

1. Environment Configuration

```yaml
# config/production.yaml
database:
  host: "localhost"
  port: 5432
  database: "gvrc_admin_production"
  username: "gvrc_user"
  password: "gvrc_password123"
  pool_size: 20
  timeout: 30

etl:
  batch_size: 100
  max_retries: 3
  timeout_seconds: 600
  enable_parallel_processing: true
  max_concurrent_tasks: 8

logging:
  level: "INFO"
  format: "json"
  file_path: "logs/etl_pipeline.log"

security:
  enable_encryption: true
  enable_audit_logging: true
  encryption_password: "${ENCRYPTION_PASSWORD}"

monitoring:
  enable_metrics: true
  metrics_port: 8080
  health_check_interval: 30
```

2. Docker Configuration

```dockerfile
# Dockerfile
FROM python:3.12-slim

WORKDIR /app

# Install system dependencies
```

```
RUN apt-get update && apt-get install -y \
    postgresql-client \
    && rm -rf /var/lib/apt/lists/*

# Copy requirements
COPY config/etl_requirements.txt .

# Install Python dependencies
RUN pip install --no-cache-dir -r etl_requirements.txt

# Copy application code
COPY src/ ./src/
COPY config/ ./config/

# Create logs directory
RUN mkdir -p logs

# Set environment variables
ENV PYTHONPATH=/app
ENV DJANGO_SETTINGS_MODULE=core.settings.production

# Expose metrics port
EXPOSE 8080

# Health check
HEALTHCHECK --interval=30s --timeout=10s --start-period=5s --retries=3 \
    CMD python -c "import requests;
requests.get('http://localhost:8080/health')"

# Run ETL pipeline
CMD ["python", "src/scripts/production_json_importer.py"]
```

3. Kubernetes Deployment

```
# k8s/etl-pipeline.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: etl-pipeline
  labels:
    app: etl-pipeline
spec:
  replicas: 2
  selector:
    matchLabels:
      app: etl-pipeline
  template:
    metadata:
      labels:
        app: etl-pipeline
    spec:
```

```
      containers:
      - name: etl-pipeline
        image: gvrc/etl-pipeline:latest
        ports:
        - containerPort: 8080
        env:
        - name: DATABASE_HOST
          value: "postgres-service"
        - name: DATABASE_PORT
          value: "5432"
        - name: DATABASE_NAME
          value: "gvrc_admin_production"
        - name: DATABASE_USER
          valueFrom:
            secretKeyRef:
              name: database-secret
              key: username
        - name: DATABASE_PASSWORD
          valueFrom:
            secretKeyRef:
              name: database-secret
              key: password
        resources:
          requests:
            memory: "512Mi"
            cpu: "250m"
          limits:
            memory: "1Gi"
            cpu: "500m"
        livenessProbe:
          httpGet:
            path: /health
            port: 8080
          initialDelaySeconds: 30
          periodSeconds: 10
        readinessProbe:
          httpGet:
            path: /ready
            port: 8080
          initialDelaySeconds: 5
          periodSeconds: 5
---
apiVersion: v1
kind: Service
metadata:
  name: etl-pipeline-service
spec:
  selector:
    app: etl-pipeline
  ports:
  - port: 8080
    targetPort: 8080
  type: ClusterIP
```

This comprehensive guide covers the essential best practices for data engineering, providing a solid foundation for building and maintaining production-ready ETL systems.