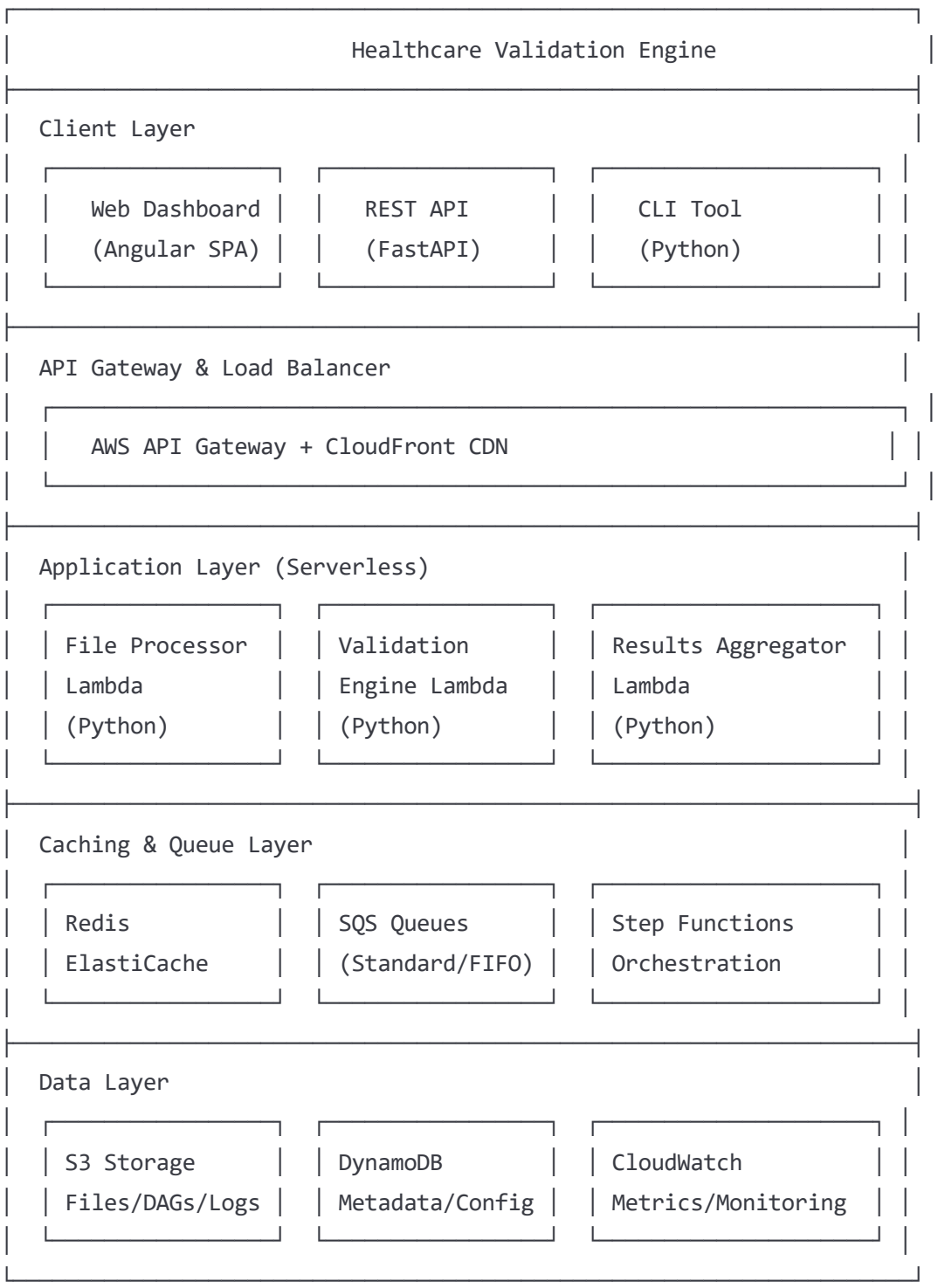
Healthcare Data Validation Rule Engine: High-Level Design Document

Document Information

- **Version:** 1.0
- **Date:** May 24, 2025
- **Classification:** Technical Design - Implementation Ready
- **Review Status:** Engineering Review Required

System Overview

Architecture Diagram



Core Components

1. File Processing Service

- Input validation and format detection
- File chunking and streaming
- Event-driven processing triggers

2. Validation Engine Service

- Graph-based rule execution
- Multi-tier caching
- Extensible validation plugins

3. Results Management Service

- Audit trail generation
- Compliance reporting
- Cost tracking and alerting

4. Configuration Management Service

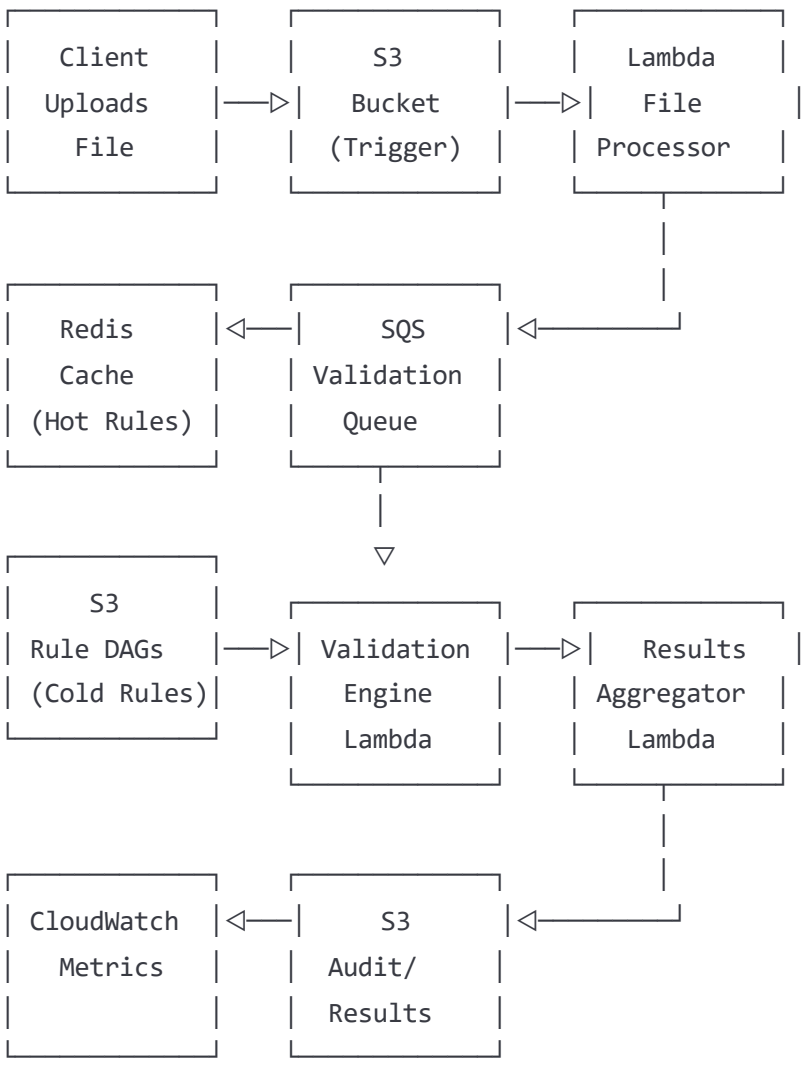
- Rule version management
- Customer settings
- Feature flag controls

Data Flow Architecture

Primary Data Flow

```
File Upload → S3 Event → File Processor → Validation Queue →
Validation Engine → Results Queue → Results Aggregator →
Audit Storage → Customer Notification
```

Detailed Flow Diagram



Interface Contracts & Data Models

Core Interface Specifications

Validation Plugin Interface

python

```python
from abc import ABC, abstractmethod
from typing import Dict, List, Optional
from dataclasses import dataclass
from enum import Enum

class ValidationTier(Enum):
    SIMPLE_FIELD = "simple_field"
    CROSS_FIELD = "cross_field"
    PATTERN_MATCH = "pattern_match"

class ValidationSeverity(Enum):
    ERROR = "error"
    WARNING = "warning"
    INFO = "info"

@dataclass
class ValidationRule:
    """Standard validation rule structure"""
    rule_id: str
    name: str
    tier: ValidationTier
    target_fields: List[str]
    dependencies: List[str]
    cms_code: str
    description: str
    error_message: str
    severity: ValidationSeverity
    enabled: bool = True
    version: str = "1.0"

@dataclass
class EncounterRecord:
    """Standard healthcare encounter record structure"""
    record_id: str
    patient_id: Optional[str]
    member_id: str
    provider_npi: str
    service_date: str
    diagnosis_codes: List[str]
    procedure_codes: List[str]
    place_of_service: str
    claim_amount: float
    insurance_type: str
```

```python
    additional_data: Dict = None

    def get_field_value(self, field_path: str) -> Optional[str]:
        """Get nested field value using dot notation (e.g., 'member.gender')"""
        fields = field_path.split('.')
        value = self.__dict__

        for field in fields:
            if isinstance(value, dict) and field in value:
                value = value[field]
            elif hasattr(value, field):
                value = getattr(value, field)
            else:
                return None
        return value

@dataclass
class ValidationResult:
    """Standard validation result structure"""
    rule_id: str
    status: str  # 'passed', 'failed', 'warning', 'skipped'
    message: str
    severity: ValidationSeverity
    field_path: Optional[str] = None
    expected_value: Optional[str] = None
    actual_value: Optional[str] = None
    cms_code: Optional[str] = None
    execution_time_ms: float = 0.0

class ValidatorPlugin(ABC):
    """Base class for all validation plugins"""

    @abstractmethod
    def can_validate(self, rule: ValidationRule) -> bool:
        """Check if this plugin can handle the given rule"""
        pass

    @abstractmethod
    async def execute(self, rule: ValidationRule, record: EncounterRecord) -> ValidationResult:
        """Execute validation rule against record"""
        pass

    @abstractmethod
    def get_supported_tiers(self) -> List[ValidationTier]:
```

```python
        """Return list of validation tiers this plugin supports"""
        pass

# Example Plugin Implementation
class SimpleFieldValidator(ValidatorPlugin):
    """Validates simple field format and value constraints"""

    def can_validate(self, rule: ValidationRule) -> bool:
        return rule.tier == ValidationTier.SIMPLE_FIELD

    def get_supported_tiers(self) -> List[ValidationTier]:
        return [ValidationTier.SIMPLE_FIELD]

    async def execute(self, rule: ValidationRule, record: EncounterRecord) -> ValidationResult:
        """Execute simple field validation"""

        start_time = time.time()

        try:
            # Get field value from record
            field_path = rule.target_fields[0] if rule.target_fields else None
            if not field_path:
                return ValidationResult(
                    rule_id=rule.rule_id,
                    status='failed',
                    message='No target field specified',
                    severity=rule.severity
                )

            actual_value = record.get_field_value(field_path)

            # Apply validation logic based on rule configuration
            validation_passed = await self._validate_field_value(
                actual_value, rule.rule_id, rule.cms_code
            )

            return ValidationResult(
                rule_id=rule.rule_id,
                status='passed' if validation_passed else 'failed',
                message=rule.error_message if not validation_passed else 'Validation passed',
                severity=rule.severity,
                field_path=field_path,
                actual_value=str(actual_value) if actual_value else None,
                cms_code=rule.cms_code,
```

```
                execution_time_ms=(time.time() - start_time) * 1000
            )

        except Exception as e:
            return ValidationResult(
                rule_id=rule.rule_id,
                status='failed',
                message=f'Validation error: {str(e)}',
                severity=ValidationSeverity.ERROR,
                execution_time_ms=(time.time() - start_time) * 1000
            )

# Sample Rule Definitions
SAMPLE_CMS_RULES = [
    {
        "rule_id": "TRC_004",
        "name": "Member Gender Validation",
        "type": "field_format",
        "tier": "simple_field",
        "target_fields": ["member.gender"],
        "expected_format": "M|F|U",
        "cms_code": "TRC004",
        "description": "Member gender must be M, F, or U",
        "error_message": "Invalid member gender code. Expected M, F, or U",
        "severity": "error"
    },
    {
        "rule_id": "TRC_015",
        "name": "Service Date Range Validation",
        "type": "cross_field",
        "tier": "cross_field",
        "target_fields": ["service_date_from", "service_date_to"],
        "dependencies": ["TRC_004"],
        "cms_code": "TRC015",
        "description": "Service date range must be valid and logical",
        "error_message": "Service date range is invalid or illogical",
        "severity": "warning"
    },
    {
        "rule_id": "TRC_025",
        "name": "Diagnosis Code Format",
        "type": "pattern_match",
        "tier": "pattern_match",
        "target_fields": ["diagnosis_codes"],
```

```
        "pattern": "^[A-Z][0-9]{2}(\\.?[0-9A-Z]{1,4})?$",
        "cms_code": "TRC025",
        "description": "ICD-10 diagnosis codes must follow standard format",
        "error_message": "Invalid ICD-10 diagnosis code format",
        "severity": "error"
    }
]
```

Plugin Registration System

python

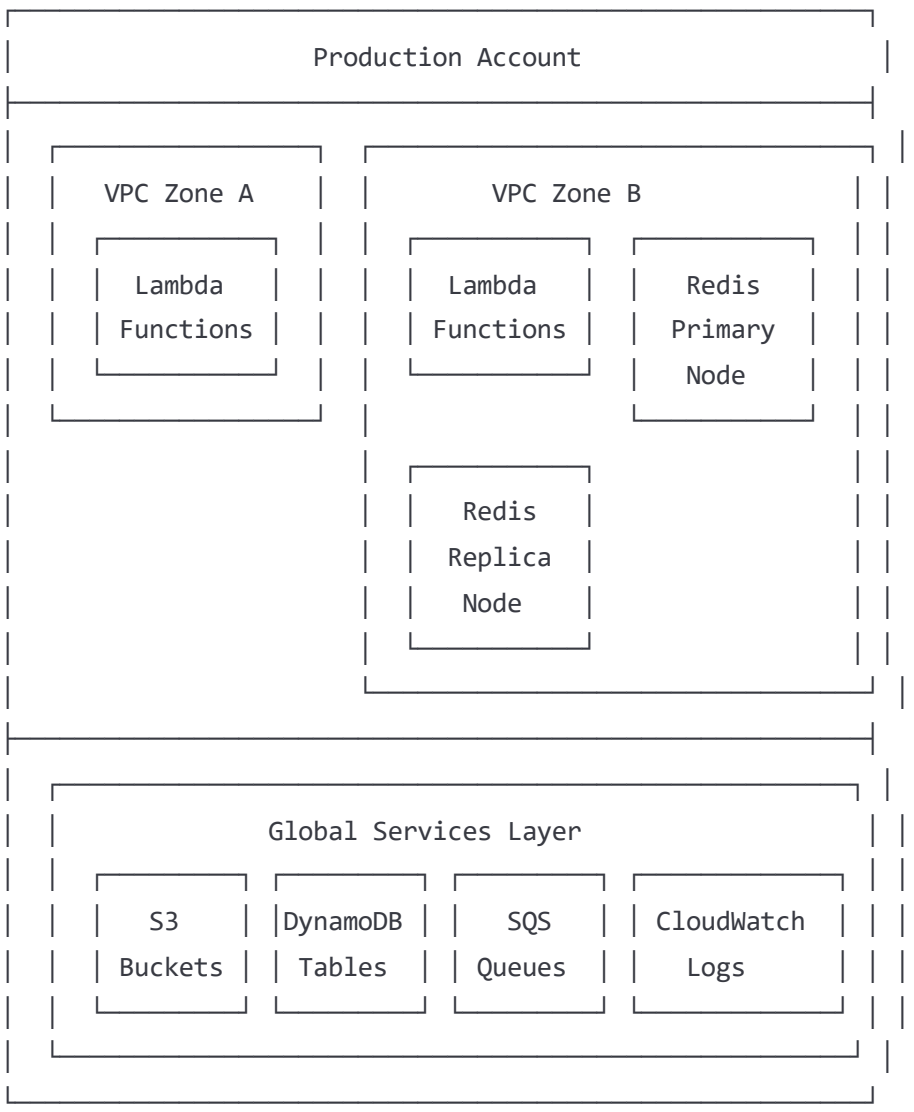
```python
class PluginRegistry:
    """Central registry for validation plugins"""

    def __init__(self):
        self.plugins: Dict[ValidationTier, List[ValidatorPlugin]] = {
            ValidationTier.SIMPLE_FIELD: [],
            ValidationTier.CROSS_FIELD: [],
            ValidationTier.PATTERN_MATCH: []
        }

    def register_plugin(self, plugin: ValidatorPlugin):
        """Register a validation plugin"""
        for tier in plugin.get_supported_tiers():
            self.plugins[tier].append(plugin)

    def get_plugin_for_rule(self, rule: ValidationRule) -> Optional[ValidatorPlugin]:
        """Get appropriate plugin for validation rule"""
        tier_plugins = self.plugins.get(rule.tier, [])

        for plugin in tier_plugins:
            if plugin.can_validate(rule):
                return plugin

        return None

# Global plugin registry
plugin_registry = PluginRegistry()

# Register default plugins
plugin_registry.register_plugin(SimpleFieldValidator())
plugin_registry.register_plugin(CrossFieldValidator())
plugin_registry.register_plugin(PatternMatchValidator())
```

Deployment Topology & Service Boundaries

Component Isolation Strategy

Component	Isolation Level	Availability	Security Boundary	Notes
Lambda Functions	Shared Account	Multi-AZ	VPC + Security Groups	Ephemeral compute, auto-scaling
Redis Cache	Shared Cluster	Multi-AZ	VPC + AUTH token	Single-AZ fallback possible
S3 Buckets	Customer Isolated	Global	Bucket policies + KMS	Encrypted at rest, lifecycle policies
DynamoDB Tables	Shared Tables	Multi-AZ	IAM + Row-level security	Point-in-time recovery enabled
SQS Queues	Shared Queues	Multi-AZ	IAM policies	Message-level customer isolation
Audit Logs	Customer Isolated	Multi-AZ	Separate KMS keys	7-year retention, compliance partition
API Gateway	Shared	Multi-AZ	JWT + rate limiting	Customer-specific throttling
CloudWatch	Shared Namespace	Multi-AZ	Log groups by customer	Cost tracking by customer tags

Service Boundary Map



Blast Radius Analysis

python

```python
# Service dependency mapping for failure analysis
SERVICE_DEPENDENCIES = {
    'file_processor_lambda': {
        'critical_dependencies': ['s3_input_bucket', 'sqs_validation_queue'],
        'optional_dependencies': ['redis_cache', 'cloudwatch_logs'],
        'blast_radius': 'single_customer_file',
        'recovery_time': '< 5 minutes'
    },
    'validation_engine_lambda': {
        'critical_dependencies': ['s3_rules_bucket', 'redis_cache'],
        'optional_dependencies': ['dynamodb_config'],
        'blast_radius': 'multiple_customers_processing',
        'recovery_time': '< 10 minutes'
    },
    'redis_cache': {
        'critical_dependencies': ['vpc_networking'],
        'optional_dependencies': ['cloudwatch_monitoring'],
        'blast_radius': 'all_customers_performance_degradation',
        'recovery_time': '< 15 minutes'
    }
}
```

Operational Playbook & Incident Response

Production Support Scenarios

Scenario 1: DAG Version Mismatch

```yaml
Trigger: ValidationResult contains "Unknown rule version" error
Symptoms:
  - Rule execution failures
  - CloudWatch errors mentioning rule version conflicts
  - Customer validation jobs failing

Immediate Actions:
  1. Check S3 for latest DAG version: aws s3 ls s3://rules-bucket/compiled-dags/
  2. Clear Redis cache for affected customer: redis-cli DEL rule_graph:customer_id:*
  3. Trigger DAG reload from S3 cold storage
  4. Monitor validation success rate for next 10 minutes

Root Cause Investigation:
  - Check GitHub Actions for failed DAG compilation
  - Verify CMS rule update deployment pipeline
  - Check for manual cache invalidations

Prevention:
  - Add DAG version health checks to monitoring
  - Implement graceful version transition periods
```

Scenario 2: Redis Cache Premature Eviction

yaml

```yaml
Trigger: Cache hit rate drops below 50% over 1 hour
Symptoms:
  - Increased S3 API calls
  - Higher Lambda execution times
  - Customer cost increases

Immediate Actions:
  1. Check Redis memory usage: redis-cli INFO memory
  2. Identify evicted keys: redis-cli LASTSAVE
  3. Trigger warm reload of hot DAGs
  4. Send SNS notification to ops team

Auto-Remediation:
  - Scale Redis instance to next size if memory > 80%
  - Implement cache warming Lambda for frequently accessed rules
  - Adjust TTL settings based on usage patterns

Monitoring:
  - Set CloudWatch alarm for cache hit rate < 70%
  - Track S3 GET request costs per customer
```

Scenario 3: Audit TTL Compliance Boundary

```yaml
Trigger: Audit logs approaching 7-year retention limit
Symptoms:
  - CloudWatch log showing TTL warnings
  - Compliance dashboard showing red status
  - Automated compliance checks failing

Immediate Actions:
  1. Identify logs nearing retention limit
  2. Check for CMS-required audit classifications
  3. Auto-extend retention for compliance-critical logs
  4. Alert compliance officer via email + Slack

Escalation Path:
  - Level 1: Auto-extend retention + notify ops
  - Level 2: Manual review of log classifications
  - Level 3: Legal/compliance team review

Prevention:
  - Implement audit log classification at ingestion
  - Add buffer period before actual TTL expiration
  - Regular compliance audit of retention policies
```

Scenario 4: Cascade Scaling Failure

```yaml
Trigger: Multiple services hitting scaling limits simultaneously
Symptoms:
  - SQS queue depth > 10K messages
  - Lambda concurrent executions at limit
  - Redis CPU > 90%
  - Customer timeouts increasing

Immediate Actions:
  1. Enable throttling on API Gateway
  2. Increase Lambda reserved concurrency temporarily
  3. Scale Redis cluster vertically
  4. Switch to "safe mode" processing (basic rules only)

Safe Mode Configuration:
  - Use only top 50 critical CMS rules
  - Reduce batch sizes to minimize Lambda memory usage
  - Disable non-essential audit logging
  - Prioritize paying customers in queue processing

Recovery Steps:
  1. Monitor queue drain rate
  2. Gradually re-enable full rule processing
  3. Restore normal audit levels
  4. Conduct post-incident review
```

Health Check & Monitoring Endpoints

python

```python
# Health check implementation
class HealthChecker:
    async def comprehensive_health_check(self) -> Dict:
        """Production-ready health check"""

        health_status = {
            'timestamp': datetime.utcnow().isoformat(),
            'overall_status': 'healthy',
            'component_status': {},
            'performance_metrics': {},
            'alerts': []
        }

        # Check critical dependencies
        checks = [
            ('redis_cache', self.check_redis_health),
            ('s3_buckets', self.check_s3_health),
            ('sqs_queues', self.check_sqs_health),
            ('lambda_functions', self.check_lambda_health),
            ('rule_graph_availability', self.check_rule_graph_health)
        ]

        for check_name, check_func in checks:
            try:
                result = await check_func()
                health_status['component_status'][check_name] = result

                if result['status'] != 'healthy':
                    health_status['overall_status'] = 'degraded'

            except Exception as e:
                health_status['component_status'][check_name] = {
                    'status': 'unhealthy',
                    'error': str(e),
                    'timestamp': datetime.utcnow().isoformat()
                }
                health_status['overall_status'] = 'unhealthy'

        return health_status

    async def check_redis_health(self) -> Dict:
        """Check Redis cache health and performance"""
```

```
    redis_client = redis.Redis.from_url(os.environ['REDIS_URL'])

    start_time = time.time()
    info = redis_client.info()
    response_time = (time.time() - start_time) * 1000

    memory_usage = info['used_memory'] / info['maxmemory']
    hit_rate = info['keyspace_hits'] / (info['keyspace_hits'] + info['keyspace_misses'])

    status = 'healthy'
    if memory_usage > 0.9:
        status = 'degraded'
    elif response_time > 100:  # ms
        status = 'degraded'
    elif hit_rate < 0.5:
        status = 'degraded'

    return {
        'status': status,
        'metrics': {
            'response_time_ms': response_time,
            'memory_usage_percent': memory_usage * 100,
            'hit_rate_percent': hit_rate * 100,
            'connected_clients': info['connected_clients']
        }
    }
```

Testing & Validation Strategy

Comprehensive Testing Matrix

Component	Test Type	Tool/Framework	Frequency	Coverage Target	Quality Gate
Validation Engine	Unit Tests	PyTest + AsyncIO	CI/CD	95% line coverage	All tests pass
Rule Accuracy	Rule Coverage Tests	Custom JSON samples	CI/CD	100% CMS rules	No false positives
DAG Compilation	Version Checks	GitHub Actions	On commit	All rule versions	Valid DAG output
Redis Cache	TTL Enforcement	Integration tests	Daily	Cache policies	No stale data
File Processing	Format Validation	Healthcare parsers	CI/CD	X12, HL7, CSV	Parse success rate >99%
API Endpoints	Contract Testing	Postman/Insomnia	CI/CD	All endpoints	Response schema valid
Security	Vulnerability Scan	OWASP ZAP	Weekly	Dependencies	No high/critical vulns
Performance	Load Testing	Locust/Artillery	Pre-release	1K concurrent users	<10s response time
Audit Logs	Schema + TTL	Athena queries	Weekly	Log structure	Compliance requirements met
Cost Tracking	Billing Accuracy	Custom scripts	Daily	Cost calculations	<5% variance from AWS

Rule Accuracy Testing Framework

python

```python
class RuleAccuracyTester:
    """Comprehensive rule testing framework"""

    def __init__(self):
        self.test_cases_db = TestCasesDatabase()
        self.validation_engine = ValidationEngine()

    async def test_rule_coverage(self) -> Dict:
        """Test coverage of all CMS rules"""

        cms_rules = await self.load_all_cms_rules()
        test_results = {}

        for rule in cms_rules:
            # Get test cases for this rule
            test_cases = await self.test_cases_db.get_test_cases(rule.rule_id)

            rule_results = {
                'rule_id': rule.rule_id,
                'test_cases_count': len(test_cases),
                'passed': 0,
                'failed': 0,
                'false_positives': 0,
                'false_negatives': 0,
                'details': []
            }

            for test_case in test_cases:
                result = await self._execute_rule_test_case(rule, test_case)
                rule_results['details'].append(result)

                if result['expected_outcome'] == result['actual_outcome']:
                    rule_results['passed'] += 1
                else:
                    rule_results['failed'] += 1

                    # Classify error type
                    if result['expected_outcome'] == 'pass' and result['actual_outcome'] == 'fa
                        rule_results['false_positives'] += 1
                    elif result['expected_outcome'] == 'fail' and result['actual_outcome'] == '
                        rule_results['false_negatives'] += 1

            test_results[rule.rule_id] = rule_results
```

```
        return {
            'overall_accuracy': self._calculate_overall_accuracy(test_results),
            'rule_results': test_results,
            'summary': self._generate_test_summary(test_results)
        }

    async def _execute_rule_test_case(self, rule: ValidationRule, test_case: Dict) -> Dict:
        """Execute single rule test case"""

        # Create test record from test case data
        test_record = EncounterRecord(**test_case['record_data'])

        # Execute validation
        start_time = time.time()
        validation_result = await self.validation_engine.validate_single_rule(rule, test_record
        execution_time = (time.time() - start_time) * 1000

        return {
            'test_case_id': test_case['id'],
            'expected_outcome': test_case['expected_outcome'],
            'actual_outcome': validation_result.status,
            'execution_time_ms': execution_time,
            'error_message': validation_result.message if validation_result.status == 'failed'
        }

# Sample test cases for rule accuracy
RULE_TEST_CASES = [
    {
        'id': 'TRC004_valid_gender',
        'rule_id': 'TRC_004',
        'description': 'Valid member gender codes',
        'expected_outcome': 'pass',
        'record_data': {
            'record_id': 'test_001',
            'member_id': 'M123456',
            'provider_npi': '1234567890',
            'service_date': '2025-05-24',
            'diagnosis_codes': ['Z23.1'],
            'procedure_codes': ['99213'],
            'place_of_service': '11',
            'claim_amount': 150.00,
            'insurance_type': 'MA',
            'additional_data': {
```

```
                'member': {'gender': 'M'}  # Valid gender code
            }
        }
    },
    {
        'id': 'TRC004_invalid_gender',
        'rule_id': 'TRC_004',
        'description': 'Invalid member gender code',
        'expected_outcome': 'fail',
        'record_data': {
            'record_id': 'test_002',
            'member_id': 'M123457',
            'provider_npi': '1234567890',
            'service_date': '2025-05-24',
            'diagnosis_codes': ['Z23.1'],
            'procedure_codes': ['99213'],
            'place_of_service': '11',
            'claim_amount': 150.00,
            'insurance_type': 'MA',
            'additional_data': {
                'member': {'gender': 'X'}  # Invalid gender code
            }
        }
    }
]
```

Performance Benchmarking Framework

python

```python
class PerformanceBenchmark:
    """Performance testing and benchmarking"""

    async def benchmark_throughput(self, record_counts: List[int]) -> Dict:
        """Benchmark processing throughput at different scales"""

        results = {}

        for record_count in record_counts:
            print(f"Benchmarking {record_count} records...")

            # Generate test data
            test_records = self.generate_test_records(record_count)

            # Execute benchmark
            start_time = time.time()
            validation_results = await self.process_records_batch(test_records)
            end_time = time.time()

            processing_time = end_time - start_time
            throughput = record_count / processing_time

            results[record_count] = {
                'processing_time_seconds': processing_time,
                'throughput_records_per_second': throughput,
                'memory_peak_mb': self.get_peak_memory_usage(),
                'cache_hit_rate': self.get_cache_hit_rate(),
                'error_rate': self.calculate_error_rate(validation_results)
            }

        return {
            'benchmark_results': results,
            'performance_summary': self._analyze_performance_trends(results)
        }

    def _analyze_performance_trends(self, results: Dict) -> Dict:
        """Analyze performance scaling characteristics"""

        record_counts = sorted(results.keys())
        throughputs = [results[count]['throughput_records_per_second'] for count in record_cour

        # Calculate scaling efficiency
        baseline_throughput = throughputs[0]
```

```
        scaling_efficiency = []

        for i, throughput in enumerate(throughputs):
            expected_throughput = baseline_throughput * (record_counts[i] / record_counts[0])
            efficiency = throughput / expected_throughput
            scaling_efficiency.append(efficiency)

        return {
            'baseline_throughput': baseline_throughput,
            'max_throughput': max(throughputs),
            'scaling_efficiency': scaling_efficiency,
            'linear_scaling': all(eff >= 0.8 for eff in scaling_efficiency),
            'recommendations': self._generate_performance_recommendations(results)
        }

# Performance targets and SLA definitions
PERFORMANCE_TARGETS = {
    'throughput_records_per_second': 800,
    'response_time_p95_ms': 10000,
    'memory_usage_max_mb': 2500,
    'cache_hit_rate_min_percent': 70,
    'error_rate_max_percent': 0.1,
    'cost_per_1000_records_max': 0.15
}
```

Service Scaling Thresholds & Circuit Breakers

1. File Processing Service

Lambda Configuration

```yaml
Runtime: Python 3.11
Memory: 1024 MB
Timeout: 15 minutes
Concurrent Executions: 100
Environment Variables:
  - S3_BUCKET_NAME
  - SQS_VALIDATION_QUEUE_URL
  - REDIS_CLUSTER_ENDPOINT
  - LOG_LEVEL
```

Core Functions

python

```python
# File processor entry point
import asyncio
import json
import boto3
from typing import Dict, List
from healthcare_parsers import X12Parser, HL7Parser

async def lambda_handler(event, context):
    """Main file processing handler"""

    s3_event = event['Records'][0]['s3']
    bucket = s3_event['bucket']['name']
    key = s3_event['object']['key']

    try:
        # Validate file format and size
        file_metadata = await validate_file(bucket, key)

        # Determine processing strategy
        strategy = determine_processing_strategy(file_metadata)

        # Chunk file if necessary
        if file_metadata['size'] > MAX_SINGLE_PROCESS_SIZE:
            await chunk_and_queue(bucket, key, strategy)
        else:
            await queue_for_processing(bucket, key, strategy)

        return {'statusCode': 200, 'body': 'File processed successfully'}
    except Exception as error:
        await handle_processing_error(error, bucket, key)
        raise error

# File chunking logic with Python healthcare libraries
async def chunk_and_queue(bucket: str, key: str, strategy: Dict):
    """Chunk healthcare files using Python parsers"""

    s3_client = boto3.client('s3')

    # Download file content
    response = await s3_client.get_object(Bucket=bucket, Key=key)
    content = response['Body'].read().decode('utf-8')

    # Parse based on file type
```

```
    if key.endswith('.x12') or key.endswith('.edi'):
        parser = X12Parser()
        records = parser.parse_encounter_data(content)
    elif key.endswith('.hl7'):
        parser = HL7Parser()
        records = parser.parse_messages(content)
    else:
        # CSV or JSON parsing
        records = await parse_structured_data(content)

    # Create chunks
    chunks = create_record_chunks(records, strategy['chunk_size'])

    # Queue each chunk for validation
    sqs_client = boto3.client('sqs')
    for chunk_index, chunk in enumerate(chunks):
        message = {
            'bucket': bucket,
            'original_key': key,
            'chunk_data': chunk,
            'chunk_index': chunk_index,
            'total_chunks': len(chunks),
            'processing_strategy': strategy
        }

        await sqs_client.send_message(
            QueueUrl=os.environ['SQS_VALIDATION_QUEUE_URL'],
            MessageBody=json.dumps(message)
        )
```

Processing Strategy Algorithm

python

```python
def determine_processing_strategy(file_metadata: Dict) -> Dict:
    """Determine optimal processing strategy for healthcare files"""

    size_mb = file_metadata['size'] / (1024 * 1024)
    record_count = file_metadata.get('estimated_record_count', 0)
    file_type = file_metadata['file_type']

    # Healthcare-specific complexity scoring
    complexity_score = calculate_healthcare_complexity(
        file_type=file_type,
        record_count=record_count,
        has_custom_rules=file_metadata.get('has_custom_rules', False)
    )

    # Calculate optimal chunk size
    base_chunk_size = 5000  # Records per chunk

    # Adjust for file complexity
    if file_type in ['x12', 'edi']:
        complexity_factor = min(complexity_score / 100, 2.0)
    else:
        complexity_factor = min(complexity_score / 150, 1.5)

    optimal_chunk_size = max(
        int(base_chunk_size / complexity_factor),
        500  # Minimum chunk size
    )

    return {
        'chunk_size': optimal_chunk_size,
        'parallelism': 'high' if record_count > 25000 else 'standard',
        'caching_strategy': 'aggressive' if complexity_score > 200 else 'standard',
        'audit_level': file_metadata.get('customer_tier', 'lite'),
        'file_type': file_type,
        'complexity_score': complexity_score
    }

def calculate_healthcare_complexity(file_type: str, record_count: int, has_custom_rules: bool)
    """Calculate processing complexity for healthcare files"""

    base_complexity = {
        'x12': 150,      # X12 EDI files are complex
        'edi': 150,      # EDI files require careful parsing
```

```
        'hl7': 120,      # HL7 messages have structured complexity
        'csv': 50,       # CSV is straightforward
        'json': 40       # JSON is simplest
    }.get(file_type, 100)

    # Scale complexity with record count
    volume_factor = min(record_count / 10000, 2.0)  # Cap at 2x

    # Add custom rule complexity
    custom_rule_factor = 1.3 if has_custom_rules else 1.0

    return int(base_complexity * volume_factor * custom_rule_factor)
```

2. Validation Engine Service

Lambda Configuration

```yaml
Runtime: Python 3.11
Memory: 3008 MB (max for cost efficiency)
Timeout: 15 minutes
Provisioned Concurrency: 10 (during business hours)
Reserved Concurrency: 50
Environment Variables:
  - REDIS_CLUSTER_ENDPOINT
  - S3_RULES_BUCKET
  - DYNAMODB_CONFIG_TABLE
  - AUDIT_S3_BUCKET
```

Core Architecture

python

```python
import asyncio
import json
from typing import Dict, List, Optional
from dataclasses import dataclass
from enum import Enum

class ValidationTier(Enum):
    SIMPLE_FIELD = "simple_field"
    CROSS_FIELD = "cross_field"
    PATTERN_MATCH = "pattern_match"

@dataclass
class ValidationRule:
    id: str
    name: str
    tier: ValidationTier
    dependencies: List[str]
    cms_code: str
    error_message: str
    severity: str

@dataclass
class ValidationContext:
    customer_id: str
    file_id: str
    chunk_id: str
    audit_level: str
    cost_tracking: Dict

class ValidationEngine:
    def __init__(self):
        self.cache_manager = CacheManager()
        self.rule_graph = None
        self.validators = {
            ValidationTier.SIMPLE_FIELD: SimpleFieldValidator(),
            ValidationTier.CROSS_FIELD: CrossFieldValidator(),
            ValidationTier.PATTERN_MATCH: PatternMatchValidator()
        }

    async def validate_records(self, records: List[Dict], context: ValidationContext):
        """Main validation entry point"""

        # Load rule graph (cached)
```

```
        self.rule_graph = await self.load_rule_graph(context.customer_id)

        # Execute validation pipeline
        results = []
        for record in records:
            record_result = await self.validate_single_record(record, context)
            results.append(record_result)

        return results

    async def validate_single_record(self, record: Dict, context: ValidationContext):
        """Validate single record using graph optimization"""

        # Get applicable rules for record type
        applicable_rules = self.rule_graph.get_applicable_rules(record)

        # Execute rules in dependency order
        validation_results = []
        for rule_cluster in self.rule_graph.get_execution_clusters(applicable_rules):
            cluster_results = await self.execute_rule_cluster(
                rule_cluster, record, context
            )
            validation_results.extend(cluster_results)

            # Short-circuit on critical failures
            if self.has_critical_failures(cluster_results):
                break

        return {
            'record_id': record.get('id'),
            'validation_results': validation_results,
            'overall_status': self.calculate_overall_status(validation_results),
            'processing_metadata': {
                'rules_executed': len(validation_results),
                'execution_time_ms': context.get('execution_time'),
                'cache_hits': context.get('cache_hits', 0)
            }
        }
```

Multi-Tier Caching Implementation

python

```python
class CacheManager:
    def __init__(self):
        self.lambda_cache = {}  # In-memory cache
        self.redis_client = None
        self.s3_client = None

    async def get_rule_graph(self, customer_id: str, version: str) -> Optional[RuleGraph]:
        """Multi-tier cache retrieval"""

        cache_key = f"rule_graph:{customer_id}:{version}"

        # Level 1: Lambda memory (hot cache)
        if cache_key in self.lambda_cache:
            return self.lambda_cache[cache_key]

        # Level 2: Redis (warm cache)
        redis_result = await self.redis_client.get(cache_key)
        if redis_result:
            rule_graph = RuleGraph.from_json(redis_result)
            # Promote to Lambda cache
            self.lambda_cache[cache_key] = rule_graph
            return rule_graph

        # Level 3: S3 (cold storage)
        s3_key = f"compiled-dags/{customer_id}/{version}/rule_graph.json"
        s3_result = await self.s3_client.get_object(
            Bucket=os.environ['S3_RULES_BUCKET'],
            Key=s3_key
        )

        if s3_result:
            rule_graph = RuleGraph.from_json(s3_result['Body'].read())

            # Populate caches
            await self.redis_client.setex(
                cache_key, 86400, rule_graph.to_json()  # 24 hour TTL
            )
            self.lambda_cache[cache_key] = rule_graph

            return rule_graph

        return None
```

```python
    async def cache_validation_result(self, rule_id: str, input_hash: str, result: Dict):
        """Cache expensive validation results"""

        cache_key = f"validation:{rule_id}:{input_hash}"

        # Cache in Redis with TTL based on rule type
        ttl = 3600 if result.get('cacheable', True) else 300  # 1 hour or 5 minutes
        await self.redis_client.setex(cache_key, ttl, json.dumps(result))
```

Rule Graph Implementation

python

```python
class RuleGraph:
    def __init__(self):
        self.nodes = {}  # rule_id -> RuleNode
        self.edges = {}  # rule_id -> [dependent_rule_ids]
        self.execution_order = []

    @classmethod
    def from_cms_rules(cls, cms_rules: List[Dict]) -> 'RuleGraph':
        """Build rule graph from CMS rule definitions"""

        graph = cls()

        # Create nodes
        for rule_data in cms_rules:
            rule = ValidationRule(
                id=rule_data['id'],
                name=rule_data['name'],
                tier=ValidationTier(rule_data['complexity_tier']),
                dependencies=rule_data.get('dependencies', []),
                cms_code=rule_data['cms_code'],
                error_message=rule_data['error_message'],
                severity=rule_data['severity']
            )
            graph.nodes[rule.id] = rule

        # Build dependency edges
        for rule in graph.nodes.values():
            graph.edges[rule.id] = rule.dependencies

        # Calculate topological order
        graph.execution_order = graph.topological_sort()

        return graph

    def get_execution_clusters(self, applicable_rules: List[str]) -> List[List[str]]:
        """Group rules into parallel execution clusters"""

        clusters = []
        remaining_rules = set(applicable_rules)
        processed_rules = set()

        while remaining_rules:
            # Find rules with no unprocessed dependencies
```

```
            ready_rules = []
            for rule_id in remaining_rules:
                dependencies = set(self.edges.get(rule_id, []))
                if dependencies.issubset(processed_rules):
                    ready_rules.append(rule_id)

            if not ready_rules:
                # Circular dependency detected
                raise ValueError("Circular dependency in rule graph")

            clusters.append(ready_rules)
            remaining_rules -= set(ready_rules)
            processed_rules.update(ready_rules)

        return clusters

    def topological_sort(self) -> List[str]:
        """Calculate optimal rule execution order"""

        in_degree = {rule_id: 0 for rule_id in self.nodes.keys()}

        # Calculate in-degrees
        for rule_id, dependencies in self.edges.items():
            for dep in dependencies:
                if dep in in_degree:
                    in_degree[rule_id] += 1

        # Kahn's algorithm
        queue = [rule_id for rule_id, degree in in_degree.items() if degree == 0]
        result = []

        while queue:
            rule_id = queue.pop(0)
            result.append(rule_id)

            # Update in-degrees of dependent rules
            for dependent in self.get_dependents(rule_id):
                in_degree[dependent] -= 1
                if in_degree[dependent] == 0:
                    queue.append(dependent)

        if len(result) != len(self.nodes):
            raise ValueError("Circular dependency detected in rule graph")
```

```
        return result
```

3. Data Models & Schema

DynamoDB Tables

Customer Configuration Table

```json
{
    "TableName": "healthcare-validation-customers",
    "KeySchema": [
        {
            "AttributeName": "customer_id",
            "KeyType": "HASH"
        }
    ],
    "AttributeDefinitions": [
        {
            "AttributeName": "customer_id",
            "AttributeType": "S"
        }
    ],
    "BillingMode": "PAY_PER_REQUEST",
    "GlobalSecondaryIndexes": [
        {
            "IndexName": "customer-tier-index",
            "KeySchema": [
                {
                    "AttributeName": "customer_tier",
                    "KeyType": "HASH"
                }
            ],
            "Projection": {
                "ProjectionType": "ALL"
            }
        }
    ]
}
```

Customer Configuration Schema

json


```
{
    "customer_id": "cust_123456",
    "organization_name": "Acme Health Plan",
    "customer_tier": "professional",
    "created_at": "2025-05-24T10:30:00Z",
    "settings": {
        "audit_level": "verbose",
        "cost_alerts": {
            "enabled": true,
            "monthly_budget": 500.00,
            "alert_thresholds": [0.5, 0.8, 0.95]
        },
        "validation_preferences": {
            "rule_set_version": "cms_2024_q2",
            "custom_rules": ["custom_rule_001", "custom_rule_002"],
            "error_handling": "continue_on_warning"
        },
        "notification_settings": {
            "email": "admin@acmehealth.com",
            "webhook_url": "https://acmehealth.com/webhooks/validation",
            "notification_events": ["validation_complete", "error_threshold_exceeded"]
        }
    },
    "usage_stats": {
        "records_processed_month": 145000,
        "cost_current_month": 87.50,
        "avg_processing_time_ms": 1250,
        "error_rate_percent": 2.3
    }
}
```


Validation Job Table

json

```
{
    "TableName": "healthcare-validation-jobs",
    "KeySchema": [
        {
            "AttributeName": "job_id",
            "KeyType": "HASH"
        }
    ],
    "AttributeDefinitions": [
        {
            "AttributeName": "job_id",
            "AttributeType": "S"
        },
        {
            "AttributeName": "customer_id",
            "AttributeType": "S"
        },
        {
            "AttributeName": "created_at",
            "AttributeType": "S"
        }
    ],
    "GlobalSecondaryIndexes": [
        {
            "IndexName": "customer-jobs-index",
            "KeySchema": [
                {
                    "AttributeName": "customer_id",
                    "KeyType": "HASH"
                },
                {
                    "AttributeName": "created_at",
                    "KeyType": "RANGE"
                }
            ]
        }
    ]
}
```

S3 Bucket Structure

```
healthcare-validation-engine/
├── input-files/
│   ├── {customer_id}/
│   │   ├── {year}/
│   │   │   ├── {month}/
│   │   │   │   └── {job_id}/
│   │   │   │       └── original_file.x12
├── processed-chunks/
│   ├── {customer_id}/
│   │   └── {job_id}/
│   │       ├── chunk_001.json
│   │       ├── chunk_002.json
│   │       └── chunk_N.json
├── rule-graphs/
│   ├── cms/
│   │   ├── 2024_q1/
│   │   ├── 2024_q2/
│   │   └── 2024_q3/
│   │       ├── rule_graph.json
│   │       ├── rule_metadata.json
│   │       └── validation_examples.json
├── results/
│   ├── {customer_id}/
│   │   └── {job_id}/
│   │       ├── validation_results.parquet
│   │       ├── error_summary.json
│   │       └── audit_trail.parquet
└── audit-logs/
    ├── year={year}/
    │   ├── month={month}/
    │   │   ├── day={day}/
    │   │   │   └── {customer_id}/
    │   │   │       └── validation_events.parquet
```

API Specifications

REST API Endpoints

Authentication & Authorization

```yaml
Base URL: https://api.healthcare-validation.com/v1
Authentication: Bearer Token (JWT)
Rate Limiting: 1000 requests/hour per customer
```

Core Endpoints

File Upload & Validation

```yaml
POST /validate
Content-Type: multipart/form-data
Authorization: Bearer {token}

Parameters:
  - file: File (required) - X12 EDI file or CSV
  - validation_options: JSON (optional)
    - audit_level: "lite" | "verbose"
    - rule_set_version: string
    - notification_webhook: URL
    - cost_limit: number (optional budget cap)

Response:
  200 OK:
    {
      "job_id": "job_12345",
      "status": "queued",
      "estimated_completion": "2025-05-24T10:35:00Z",
      "estimated_cost": 12.50,
      "records_detected": 5000,
      "rules_to_apply": 127
    }

  400 Bad Request:
    {
      "error": "invalid_file_format",
      "message": "File must be X12 EDI or CSV format",
      "supported_formats": ["x12", "csv", "json"]
    }
```

Job Status & Results

yaml

```
GET /jobs/{job_id}
Authorization: Bearer {token}

Response:
  200 OK:
    {
      "job_id": "job_12345",
      "status": "completed" | "processing" | "failed" | "queued",
      "progress": {
        "records_processed": 4800,
        "total_records": 5000,
        "percent_complete": 96.0
      },
      "results": {
        "validation_summary": {
          "total_records": 5000,
          "passed": 4750,
          "warnings": 200,
          "errors": 50,
          "error_rate": 1.0
        },
        "cost_breakdown": {
          "processing_cost": 2.50,
          "storage_cost": 0.25,
          "total_cost": 2.75
        },
        "download_urls": {
          "results_file": "https://s3.amazonaws.com/results/...",
          "audit_trail": "https://s3.amazonaws.com/audit/...",
          "error_report": "https://s3.amazonaws.com/errors/..."
        }
      }
    }
```

Customer Analytics

yaml

```yaml
GET /analytics/usage
Authorization: Bearer {token}
Query Parameters:
  - start_date: ISO date
  - end_date: ISO date
  - granularity: "day" | "week" | "month"

Response:
  200 OK:
    {
      "usage_summary": {
        "total_records_processed": 150000,
        "total_cost": 75.50,
        "avg_cost_per_1000_records": 0.503,
        "jobs_completed": 23,
        "avg_processing_time_minutes": 8.5
      },
      "usage_timeline": [
        {
          "date": "2025-05-20",
          "records_processed": 12000,
          "cost": 6.25,
          "error_rate": 1.8
        }
      ],
      "cost_projection": {
        "current_month_estimate": 95.00,
        "trend": "increasing",
        "budget_utilization": 0.63
      }
    }
```

WebSocket API for Real-Time Updates

```yaml
WebSocket URL: wss://ws.healthcare-validation.com/v1/jobs/{job_id}
Authentication: Query parameter ?token={jwt_token}

Message Types:
  - job_progress: Real-time processing updates
  - cost_alert: Budget threshold notifications
  - validation_error: Critical error notifications
  - job_complete: Final results notification

Example Messages:
  {
    "type": "job_progress",
    "job_id": "job_12345",
    "data": {
      "records_processed": 2500,
      "total_records": 5000,
      "current_cost": 1.25,
      "estimated_total_cost": 2.50,
      "errors_found": 12
    }
  }
```

Security & Compliance Architecture

HIPAA Compliance Implementation

Data Encryption

```yaml
Encryption at Rest:
  - S3: AES-256 with AWS KMS customer managed keys
  - DynamoDB: AWS managed encryption
  - Redis: In-transit encryption with AUTH
  - Lambda: Environment variables encrypted with KMS

Encryption in Transit:
  - All API calls: TLS 1.3
  - Internal service communication: VPC with TLS
  - Client uploads: HTTPS with certificate pinning
  - Database connections: SSL/TLS required
```

Access Control & Authentication

```yaml
Authentication:
  - JWT tokens with RS256 signing
  - Token expiration: 24 hours
  - Refresh token rotation: 7 days
  - Multi-factor authentication for admin accounts

Authorization:
  - Role-based access control (RBAC)
  - Customer data isolation (tenant-level)
  - API rate limiting by customer tier
  - Resource-level permissions

IAM Roles:
  - Lambda execution roles (least privilege)
  - Cross-service communication roles
  - Customer-specific S3 access policies
  - CloudWatch logging permissions
```

Audit Logging Architecture

python

```python
class AuditLogger:
    def __init__(self):
        self.cloudwatch_client = boto3.client('logs')
        self.s3_client = boto3.client('s3')

    async def log_data_access(self, event_data: Dict):
        """Log all PHI access events"""

        audit_event = {
            'timestamp': datetime.utcnow().isoformat(),
            'event_type': 'data_access',
            'customer_id': event_data['customer_id'],
            'user_id': event_data.get('user_id'),
            'resource_accessed': event_data['resource'],
            'action': event_data['action'],
            'ip_address': event_data.get('ip_address'),
            'user_agent': event_data.get('user_agent'),
            'request_id': event_data['request_id'],
            'compliance_flags': {
                'contains_phi': event_data.get('contains_phi', False),
                'minimum_necessary': event_data.get('minimum_necessary', True),
                'authorized_purpose': event_data.get('authorized_purpose')
            }
        }

        # Send to CloudWatch for real-time monitoring
        await self.cloudwatch_client.put_log_events(
            logGroupName='/healthcare-validation/audit',
            logStreamName=f"{event_data['customer_id']}/{datetime.utcnow().strftime('%Y/%m/%d')
            logEvents=[{
                'timestamp': int(time.time() * 1000),
                'message': json.dumps(audit_event)
            }]
        )

        # Store in S3 for long-term compliance (7 year retention)
        s3_key = f"audit-logs/year={datetime.utcnow().year}/month={datetime.utcnow().month}/day

        await self.s3_client.put_object(
            Bucket=os.environ['AUDIT_S3_BUCKET'],
            Key=s3_key,
            Body=json.dumps(audit_event),
            ServerSideEncryption='aws:kms',
```

```
            SSEKMSKeyId=os.environ['AUDIT_KMS_KEY_ID'],
            Metadata={
                'retention-years': '7',
                'data-classification': 'audit-log',
                'customer-id': event_data['customer_id']
            }
        )
```

Data Loss Prevention

python

```python
class DataProtectionService:
    def __init__(self):
        self.phi_patterns = [
            r'\b\d{3}-\d{2}-\d{4}\b',  # SSN
            r'\b\d{10}\b',              # Phone numbers
            r'\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,}\b'  # Email
        ]

    async def scan_for_phi(self, content: str) -> Dict:
        """Scan content for potential PHI"""

        findings = []
        for pattern in self.phi_patterns:
            matches = re.findall(pattern, content)
            if matches:
                findings.append({
                    'pattern': pattern,
                    'matches_count': len(matches),
                    'confidence': 'high'
                })

        return {
            'phi_detected': len(findings) > 0,
            'findings': findings,
            'scan_timestamp': datetime.utcnow().isoformat()
        }

    async def anonymize_audit_data(self, audit_data: Dict) -> Dict:
        """Remove or hash PHI from audit logs"""

        anonymized = audit_data.copy()

        # Hash potentially identifying information
        if 'ip_address' in anonymized:
            anonymized['ip_address_hash'] = hashlib.sha256(
                anonymized['ip_address'].encode()
            ).hexdigest()[:16]
            del anonymized['ip_address']

        return anonymized
```

Performance & Scaling Specifications

Lambda Function Scaling Configuration

```yaml
File Processor Lambda:
  Memory: 1024 MB
  Timeout: 15 minutes
  Reserved Concurrency: 100
  Provisioned Concurrency: 0 (cost optimization)
  Dead Letter Queue: Enabled
  Retry Configuration:
    Maximum Retry Attempts: 3
    Maximum Record Age: 24 hours

Validation Engine Lambda:
  Memory: 3008 MB (maximum)
  Timeout: 15 minutes
  Reserved Concurrency: 50
  Provisioned Concurrency: 10 (business hours: 8AM-8PM EST)
  Dead Letter Queue: Enabled
  Environment Variables:
    JAVA_TOOL_OPTIONS: "-XX:+TieredCompilation -XX:TieredStopAtLevel=1"

Results Aggregator Lambda:
  Memory: 512 MB
  Timeout: 5 minutes
  Reserved Concurrency: 25
  Provisioned Concurrency: 0
```

Auto-Scaling Policies

```yaml
SQS Queue Configuration:
  Validation Queue:
    Visibility Timeout: 900 seconds (15 minutes)
    Message Retention: 14 days
    Dead Letter Queue: After 3 failures
    Batch Size: 10 messages
    Scaling Metric: ApproximateNumberOfMessages
    Target Value: 100 messages

  Results Queue:
    Visibility Timeout: 300 seconds (5 minutes)
    Message Retention: 7 days
    Batch Size: 10 messages

Redis ElastiCache Scaling:
  Node Type: cache.t4g.micro (baseline)
  Auto Scaling:
    Min Capacity: 1 node
    Max Capacity: 3 nodes
    Target CPU Utilization: 70%
    Scale-out Cooldown: 300 seconds
    Scale-in Cooldown: 300 seconds
```

Performance Monitoring

python

```python
class PerformanceMonitor:
    def __init__(self):
        self.cloudwatch = boto3.client('cloudwatch')

    async def track_processing_metrics(self, job_data: Dict):
        """Track key performance indicators"""

        metrics = [
            {
                'MetricName': 'RecordsProcessedPerSecond',
                'Value': job_data['records_processed'] / job_data['processing_time_seconds'],
                'Unit': 'Count/Second',
                'Dimensions': [
                    {'Name': 'CustomerTier', 'Value': job_data['customer_tier']},
                    {'Name': 'FileType', 'Value': job_data['file_type']}
                ]
            },
            {
                'MetricName': 'ValidationLatencyP95',
                'Value': job_data['latency_p95_ms'],
                'Unit': 'Milliseconds'
            },
            {
                'MetricName': 'ErrorRate',
                'Value': job_data['error_rate_percent'],
                'Unit': 'Percent'
            },
            {
                'MetricName': 'CostPerThousandRecords',
                'Value': (job_data['total_cost'] / job_data['records_processed']) * 1000,
                'Unit': 'None'
            }
        ]

        await self.cloudwatch.put_metric_data(
            Namespace='HealthcareValidation/Performance',
            MetricData=metrics
        )

    async def check_sla_compliance(self, job_data: Dict) -> Dict:
        """Monitor SLA compliance"""

        sla_targets = {
```

```
        'processing_time_minutes': 30,
        'error_rate_percent': 5.0,
        'availability_percent': 99.9
    }

    compliance_status = {}
    for metric, target in sla_targets.items():
        actual_value = job_data.get(metric, 0)
        compliance_status[metric] = {
            'target': target,
            'actual': actual_value,
            'compliant': actual_value <= target if 'rate' in metric or 'time' in metric els
        }

    return compliance_status
```

Cost Optimization Architecture

Cost Tracking Implementation

python

```python
class CostTracker:
    def __init__(self):
        self.pricing = {
            'lambda_invocation': 0.0000002,  # Per invocation
            'lambda_gb_second': 0.0000166667,  # Per GB-second
            'sqs_request': 0.0000004,        # Per request
            's3_put_request': 0.0005,        # Per 1000 requests
            's3_get_request': 0.0004,        # Per 1000 requests
            's3_storage_gb_month': 0.023,    # Per GB per month
            'redis_node_hour': 0.017,        # cache.t4g.micro per hour
            'dynamodb_read_unit': 0.00025,   # Per read unit
            'dynamodb_write_unit': 0.00125   # Per write unit
        }

    async def calculate_job_cost(self, job_metrics: Dict) -> Dict:
        """Calculate actual cost for a validation job"""

        lambda_cost = (
            job_metrics['lambda_invocations'] * self.pricing['lambda_invocation'] +
            job_metrics['lambda_gb_seconds'] * self.pricing['lambda_gb_second']
        )

        storage_cost = (
            job_metrics['s3_put_requests'] * self.pricing['s3_put_request'] / 1000 +
            job_metrics['s3_get_requests'] * self.pricing['s3_get_request'] / 1000 +
            job_metrics['s3_storage_gb'] * self.pricing['s3_storage_gb_month'] / 30  # Daily ra
        )

        queue_cost = job_metrics['sqs_requests'] * self.pricing['sqs_request']

        # Redis cost allocated based on usage time
        redis_cost = (job_metrics['processing_time_hours'] *
                      self.pricing['redis_node_hour'] /
                      job_metrics['concurrent_jobs_on_redis'])

        database_cost = (
            job_metrics['dynamodb_read_units'] * self.pricing['dynamodb_read_unit'] +
            job_metrics['dynamodb_write_units'] * self.pricing['dynamodb_write_unit']
        )

        total_cost = lambda_cost + storage_cost + queue_cost + redis_cost + database_cost

        return {
```

```
            'total_cost': round(total_cost, 4),
            'breakdown': {
                'lambda': round(lambda_cost, 4),
                'storage': round(storage_cost, 4),
                'queue': round(queue_cost, 4),
                'redis': round(redis_cost, 4),
                'database': round(database_cost, 4)
            },
            'cost_per_1000_records': round((total_cost / job_metrics['records_processed']) * 10
        }

    async def optimize_cost_allocation(self, customer_usage: Dict) -> Dict:
        """Suggest cost optimizations based on usage patterns"""

        optimizations = []

        # Redis optimization
        if customer_usage['redis_hit_rate'] < 0.5:
            optimizations.append({
                'type': 'redis_optimization',
                'suggestion': 'Reduce Redis cache size or TTL',
                'potential_savings': customer_usage['redis_monthly_cost'] * 0.3
            })

        # Batch size optimization
        if customer_usage['avg_batch_size'] < 1000:
            optimizations.append({
                'type': 'batch_optimization',
                'suggestion': 'Increase batch size to reduce Lambda invocations',
                'potential_savings': customer_usage['lambda_monthly_cost'] * 0.2
            })

        # Storage lifecycle optimization
        if customer_usage['old_data_gb'] > 10:
            optimizations.append({
                'type': 'storage_lifecycle',
                'suggestion': 'Move old audit data to S3 Glacier',
                'potential_savings': customer_usage['old_data_gb'] * 0.004 * 12  # Annual savir
            })

        return {
            'current_monthly_cost': customer_usage['total_monthly_cost'],
            'optimization_suggestions': optimizations,
```

```
            'total_potential_savings': sum(opt['potential_savings'] for opt in optimizations)
        }
```

Budget Alerts & Controls

python

```python
class BudgetController:
    def __init__(self):
        self.sns_client = boto3.client('sns')

    async def check_budget_alerts(self, customer_id: str, current_cost: float):
        """Check and send budget alerts"""

        customer_config = await self.get_customer_config(customer_id)
        budget = customer_config['settings']['cost_alerts']['monthly_budget']
        thresholds = customer_config['settings']['cost_alerts']['alert_thresholds']

        utilization = current_cost / budget

        for threshold in thresholds:
            if utilization >= threshold and not self.alert_sent(customer_id, threshold):
                await self.send_budget_alert(customer_id, current_cost, budget, threshold)
                await self.mark_alert_sent(customer_id, threshold)

        # Hard limit enforcement
        if utilization >= 1.0 and customer_config['settings'].get('enforce_budget_limit', False
            await self.suspend_customer_processing(customer_id)

    async def send_budget_alert(self, customer_id: str, current_cost: float,
                                budget: float, threshold: float):
        """Send budget alert notification"""

        message = {
            'customer_id': customer_id,
            'alert_type': 'budget_threshold',
            'current_cost': current_cost,
            'monthly_budget': budget,
            'utilization_percent': (current_cost / budget) * 100,
            'threshold_percent': threshold * 100,
            'recommendations': await self.get_cost_optimization_suggestions(customer_id)
        }

        await self.sns_client.publish(
            TopicArn=f"arn:aws:sns:us-east-1:123456789012:budget-alerts-{customer_id}",
            Message=json.dumps(message),
            Subject=f"Budget Alert: {threshold*100}% of monthly limit reached"
        )
```

Monitoring & Observability

CloudWatch Dashboard Configuration

```yaml
Dashboard Name: Healthcare Validation Engine - Operational
Widgets:
  - System Health:
      - Lambda Error Rate (all functions)
      - SQS Queue Depth
      - Redis Cache Hit Rate
      - DynamoDB Throttling Events

  - Performance Metrics:
      - Records Processed Per Second
      - Average Processing Latency (P50, P95, P99)
      - Validation Error Rate
      - Cost Per 1000 Records

  - Customer Experience:
      - API Response Times
      - Job Completion Rate
      - Customer Error Reports
      - Support Ticket Volume

  - Cost Management:
      - Daily AWS Spend
      - Cost Per Customer Segment
      - Budget Utilization Alerts
      - Resource Utilization Efficiency

Alarms:
  - Lambda Error Rate > 5%
  - SQS Queue Depth > 1000 messages for 5 minutes
  - Redis CPU > 80% for 10 minutes
  - API Latency P95 > 10 seconds
  - Daily Cost > $1000 (operational alert)
```

Distributed Tracing Implementation

python

```python
import aws_xray_sdk.core
from aws_xray_sdk.core import xray_recorder, patch_all

# Patch all AWS SDK calls for tracing
patch_all()

class TracingService:
    @xray_recorder.capture('validate_file_processing')
    async def process_validation_job(self, job_data: Dict):
        """Main validation processing with distributed tracing"""

        # Create subsegment for file processing
        with xray_recorder.in_subsegment('file_processing'):
            xray_recorder.current_subsegment().put_annotation('customer_id', job_data['customer
            xray_recorder.current_subsegment().put_annotation('file_size_mb', job_data['file_si
            xray_recorder.current_subsegment().put_annotation('record_count', job_data['record_

            file_chunks = await self.chunk_file(job_data)

        # Create subsegment for rule graph loading
        with xray_recorder.in_subsegment('rule_graph_loading'):
            rule_graph = await self.load_rule_graph(job_data['customer_id'])
            xray_recorder.current_subsegment().put_metadata('rule_count', len(rule_graph.nodes)

        # Create subsegment for validation execution
        with xray_recorder.in_subsegment('validation_execution'):
            results = []
            for chunk in file_chunks:
                chunk_result = await self.validate_chunk(chunk, rule_graph)
                results.append(chunk_result)

            xray_recorder.current_subsegment().put_annotation('chunks_processed', len(results))

        return results

    @xray_recorder.capture('cache_operation')
    async def get_from_cache(self, cache_key: str, cache_type: str):
        """Traced cache operations"""

        xray_recorder.current_subsegment().put_annotation('cache_type', cache_type)
        xray_recorder.current_subsegment().put_annotation('cache_key_hash',
                                                          hashlib.md5(cache_key.encode()).hexdige
```

```
        start_time = time.time()
        result = await self.cache_manager.get(cache_key, cache_type)
        end_time = time.time()

        xray_recorder.current_subsegment().put_metadata('cache_hit', result is not None)
        xray_recorder.current_subsegment().put_metadata('cache_latency_ms', (end_time - start_t

        return result
```

Custom Metrics & Alerting

python

```python
class MetricsCollector:
    def __init__(self):
        self.cloudwatch = boto3.client('cloudwatch')

    async def collect_business_metrics(self, job_result: Dict):
        """Collect business-specific metrics"""

        # Healthcare-specific metrics
        cms_compliance_metrics = [
            {
                'MetricName': 'CMSAcceptanceRate',
                'Value': job_result['cms_acceptance_rate'],
                'Unit': 'Percent',
                'Dimensions': [
                    {'Name': 'CustomerTier', 'Value': job_result['customer_tier']},
                    {'Name': 'EncounterType', 'Value': job_result['encounter_type']}
                ]
            },
            {
                'MetricName': 'ValidationRulesCovered',
                'Value': job_result['rules_executed'],
                'Unit': 'Count'
            },
            {
                'MetricName': 'RevenueAtRisk',
                'Value': job_result['flagged_claims_dollars'],
                'Unit': 'None'  # Dollar amount
            }
        ]

        # Cost efficiency metrics
        efficiency_metrics = [
            {
                'MetricName': 'CostPerRecord',
                'Value': job_result['total_cost'] / job_result['record_count'],
                'Unit': 'None'
            },
            {
                'MetricName': 'ProcessingEfficiency',
                'Value': job_result['record_count'] / job_result['processing_time_seconds'],
                'Unit': 'Count/Second'
            }
        ]
```

```
    all_metrics = cms_compliance_metrics + efficiency_metrics

    await self.cloudwatch.put_metric_data(
        Namespace='HealthcareValidation/Business',
        MetricData=all_metrics
    )

async def create_custom_alarms(self):
    """Create business-critical alarms"""

    alarms = [
        {
            'AlarmName': 'CMS-Acceptance-Rate-Low',
            'ComparisonOperator': 'LessThanThreshold',
            'EvaluationPeriods': 2,
            'MetricName': 'CMSAcceptanceRate',
            'Namespace': 'HealthcareValidation/Business',
            'Period': 3600,  # 1 hour
            'Statistic': 'Average',
            'Threshold': 95.0,  # Alert if acceptance rate drops below 95%
            'ActionsEnabled': True,
            'AlarmActions': [
                'arn:aws:sns:us-east-1:123456789012:cms-compliance-alerts'
            ],
            'AlarmDescription': 'CMS acceptance rate has dropped below acceptable threshold
        },
        {
            'AlarmName': 'Cost-Efficiency-Degraded',
            'ComparisonOperator': 'GreaterThanThreshold',
            'EvaluationPeriods': 3,
            'MetricName': 'CostPerRecord',
            'Namespace': 'HealthcareValidation/Business',
            'Period': 3600,
            'Statistic': 'Average',
            'Threshold': 0.0015,  # Alert if cost per record exceeds $0.0015
            'ActionsEnabled': True,
            'AlarmActions': [
                'arn:aws:sns:us-east-1:123456789012:cost-optimization-alerts'
            ]
        }
    ]
```

```
        for alarm in alarms:
            await self.cloudwatch.put_metric_alarm(**alarm)
```

Deployment Architecture

Infrastructure as Code (Terraform)

hcl

```
# terraform/main.tf
terraform {
  required_version = ">= 1.0"
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 5.0"
    }
  }
}

provider "aws" {
  region = var.aws_region

  default_tags {
    tags = {
      Project     = "healthcare-validation-engine"
      Environment = var.environment
      ManagedBy   = "terraform"
    }
  }
}

# S3 Buckets
resource "aws_s3_bucket" "validation_data" {
  bucket = "${var.project_name}-validation-data-${var.environment}"
}

resource "aws_s3_bucket_versioning" "validation_data_versioning" {
  bucket = aws_s3_bucket.validation_data.id
  versioning_configuration {
    status = "Enabled"
  }
}

resource "aws_s3_bucket_encryption" "validation_data_encryption" {
  bucket = aws_s3_bucket.validation_data.id

  server_side_encryption_configuration {
    rule {
      apply_server_side_encryption_by_default {
        kms_master_key_id = aws_kms_key.s3_encryption.arn
        sse_algorithm     = "aws:kms"
```

```
      }
    }
  }
}

# Lambda Functions
resource "aws_lambda_function" "file_processor" {
  filename         = "../dist/file-processor.zip"
  function_name    = "${var.project_name}-file-processor-${var.environment}"
  role             = aws_iam_role.lambda_execution_role.arn
  handler          = "main.lambda_handler"
  runtime          = "python3.11"
  memory_size      = 1024
  timeout          = 900

  environment {
    variables = {
      S3_BUCKET_NAME          = aws_s3_bucket.validation_data.bucket
      SQS_VALIDATION_QUEUE_URL = aws_sqs_queue.validation_queue.url
      REDIS_CLUSTER_ENDPOINT   = aws_elasticache_replication_group.redis_cache.primary_endpoint
      LOG_LEVEL               = var.log_level
    }
  }

  dead_letter_config {
    target_arn = aws_sqs_queue.dlq_file_processor.arn
  }
}

resource "aws_lambda_function" "validation_engine" {
  filename         = "../dist/validation-engine.zip"
  function_name    = "${var.project_name}-validation-engine-${var.environment}"
  role             = aws_iam_role.lambda_execution_role.arn
  handler          = "main.handler"
  runtime          = "python3.11"
  memory_size      = 3008
  timeout          = 900

  reserved_concurrent_executions = 50

  environment {
    variables = {
      REDIS_CLUSTER_ENDPOINT = aws_elasticache_replication_group.redis_cache.primary_endpoint_a
      S3_RULES_BUCKET       = aws_s3_bucket.validation_data.bucket
```

```
      DYNAMODB_CONFIG_TABLE = aws_dynamodb_table.customer_config.name
      AUDIT_S3_BUCKET      = aws_s3_bucket.validation_data.bucket
    }
  }
}

# SQS Queues
resource "aws_sqs_queue" "validation_queue" {
  name                       = "${var.project_name}-validation-queue-${var.environment}"
  visibility_timeout_seconds = 900
  message_retention_seconds  = 1209600  # 14 days

  redrive_policy = jsonencode({
    deadLetterTargetArn = aws_sqs_queue.dlq_validation.arn
    maxReceiveCount     = 3
  })
}

# ElastiCache Redis
resource "aws_elasticache_replication_group" "redis_cache" {
  replication_group_id       = "${var.project_name}-cache-${var.environment}"
  description                = "Redis cache for validation rules"

  node_type                  = "cache.t4g.micro"
  port                       = 6379
  parameter_group_name       = "default.redis7"

  num_cache_clusters         = 2
  automatic_failover_enabled = true
  multi_az_enabled           = true

  subnet_group_name = aws_elasticache_subnet_group.redis_subnet_group.name
  security_group_ids = [aws_security_group.redis_sg.id]

  at_rest_encryption_enabled = true
  transit_encryption_enabled = true
  auth_token                 = var.redis_auth_token
}

# DynamoDB Tables
resource "aws_dynamodb_table" "customer_config" {
  name           = "${var.project_name}-customers-${var.environment}"
  billing_mode   = "PAY_PER_REQUEST"
  hash_key       = "customer_id"
```

```
  attribute {
    name = "customer_id"
    type = "S"
  }

  attribute {
    name = "customer_tier"
    type = "S"
  }

  global_secondary_index {
    name     = "customer-tier-index"
    hash_key = "customer_tier"
    projection_type = "ALL"
  }

  server_side_encryption {
    enabled = true
  }

  point_in_time_recovery {
    enabled = true
  }
}
```

CI/CD Pipeline (GitHub Actions)

yaml

```yaml
# .github/workflows/deploy.yml
name: Deploy Healthcare Validation Engine

on:
  push:
    branches: [main, develop]
  pull_request:
    branches: [main]

env:
  AWS_REGION: us-east-1
  TERRAFORM_VERSION: 1.5.0

jobs:
  test:
    runs-on: ubuntu-latest

    services:
      redis:
        image: redis:7
        options: >-
          --health-cmd "redis-cli ping"
          --health-interval 10s
          --health-timeout 5s
          --health-retries 5
        ports:
          - 6379:6379

    steps:
    - uses: actions/checkout@v3

    - name: Set up Python 3.11
      uses: actions/setup-python@v4
      with:
        python-version: '3.11'

    - name: Install dependencies
      run: |
        python -m pip install --upgrade pip
        pip install -r requirements.txt
        pip install -r test-requirements.txt

    - name: Run unit tests
```

```yaml
    run: |
      python -m pytest tests/unit/ -v --cov=src --cov-report=xml

  - name: Run integration tests
    run: |
      python -m pytest tests/integration/ -v
    env:
      REDIS_URL: redis://localhost:6379

  - name: Upload coverage reports
    uses: codecov/codecov-action@v3
    with:
      file: ./coverage.xml

security-scan:
  runs-on: ubuntu-latest
  steps:
  - uses: actions/checkout@v3

  - name: Run security scan
    uses: securecodewarrior/github-action-add-sarif@v1
    with:
      sarif-file: 'security-scan-results.sarif'

  - name: Dependency vulnerability scan
    run: |
      pip install safety
      safety check -r requirements.txt --json --output safety-report.json

build:
  needs: [test, security-scan]
  runs-on: ubuntu-latest

  steps:
  - uses: actions/checkout@v3

  - name: Build Lambda packages
    run: |
      # Build file processor (Python)
      cd src/file-processor
      pip install -r requirements.txt -t ./package
      cd package && zip -r ../../dist/file-processor.zip .
      cd .. && zip -g ../dist/file-processor.zip *.py
```

```
      # Build validation engine (Python)
      cd ../validation-engine
      pip install -r requirements.txt -t ./package
      cd package && zip -r ../../../dist/validation-engine.zip .
      cd .. && zip -g ../../dist/validation-engine.zip *.py

  - name: Upload build artifacts
    uses: actions/upload-artifact@v3
    with:
      name: lambda-packages
      path: dist/

deploy-staging:
  if: github.ref == 'refs/heads/develop'
  needs: build
  runs-on: ubuntu-latest
  environment: staging

  steps:
  - uses: actions/checkout@v3

  - name: Download build artifacts
    uses: actions/download-artifact@v3
    with:
      name: lambda-packages
      path: dist/

  - name: Setup Terraform
    uses: hashicorp/setup-terraform@v2
    with:
      terraform_version: ${{ env.TERRAFORM_VERSION }}

  - name: Terraform Init
    run: terraform init
    working-directory: terraform/
    env:
      AWS_ACCESS_KEY_ID: ${{ secrets.AWS_ACCESS_KEY_ID }}
      AWS_SECRET_ACCESS_KEY: ${{ secrets.AWS_SECRET_ACCESS_KEY }}

  - name: Terraform Plan
    run: terraform plan -var-file="staging.tfvars" -out=tfplan
    working-directory: terraform/

  - name: Terraform Apply
```

```
    run: terraform apply tfplan
    working-directory: terraform/

  - name: Run smoke tests
    run: |
      python -m pytest tests/smoke/ -v --environment=staging
    env:
      API_BASE_URL: ${{ steps.terraform.outputs.api_gateway_url }}

deploy-production:
  if: github.ref == 'refs/heads/main'
  needs: build
  runs-on: ubuntu-latest
  environment: production

  steps:
  - uses: actions/checkout@v3

  - name: Download build artifacts
    uses: actions/download-artifact@v3
    with:
      name: lambda-packages
      path: dist/

  - name: Setup Terraform
    uses: hashicorp/setup-terraform@v2
    with:
      terraform_version: ${{ env.TERRAFORM_VERSION }}

  - name: Terraform Apply Production
    run: |
      terraform init
      terraform plan -var-file="production.tfvars" -out=tfplan
      terraform apply tfplan
    working-directory: terraform/
    env:
      AWS_ACCESS_KEY_ID: ${{ secrets.AWS_ACCESS_KEY_ID_PROD }}
      AWS_SECRET_ACCESS_KEY: ${{ secrets.AWS_SECRET_ACCESS_KEY_PROD }}

  - name: Run production health checks
    run: |
      python scripts/health_check.py --environment=production

  - name: Notify deployment success
```

```
      uses: 8398a7/action-slack@v3
      with:
        status: success
        text: 'Healthcare Validation Engine deployed to production successfully!'
      env:
        SLACK_WEBHOOK_URL: ${{ secrets.SLACK_WEBHOOK_URL }}
```

Testing Strategy

Unit Testing Framework

python

```python
# tests/test_validation_engine.py
import pytest
from unittest.mock import AsyncMock, patch
from src.validation_engine import ValidationEngine, ValidationRule, ValidationTier

class TestValidationEngine:
    @pytest.fixture
    async def validation_engine(self):
        engine = ValidationEngine()
        await engine.initialize()
        return engine

    @pytest.fixture
    def sample_cms_rules(self):
        return [
            ValidationRule(
                id="TRC004",
                name="Diagnosis Code Format Validation",
                tier=ValidationTier.SIMPLE_FIELD,
                dependencies=[],
                cms_code="TRC004",
                error_message="Invalid diagnosis code format",
                severity="error"
            ),
            ValidationRule(
                id="TRC015",
                name="Service Date Range Validation",
                tier=ValidationTier.CROSS_FIELD,
                dependencies=["TRC004"],
                cms_code="TRC015",
                error_message="Service date range invalid",
                severity="warning"
            )
        ]

    @pytest.mark.asyncio
    async def test_simple_field_validation(self, validation_engine, sample_cms_rules):
        """Test simple field validation logic"""

        # Mock rule graph
        with patch.object(validation_engine, 'load_rule_graph') as mock_load:
            mock_rule_graph = AsyncMock()
            mock_rule_graph.get_applicable_rules.return_value = [sample_cms_rules[0]]
```

```
        mock_rule_graph.get_execution_clusters.return_value = [[sample_cms_rules[0].id]]
        mock_load.return_value = mock_rule_graph

        # Test data
        test_record = {
            'id': 'test_001',
            'diagnosis_code': 'Z23.1',  # Valid ICD-10 format
            'service_date': '2025-05-24'
        }

        context = ValidationContext(
            customer_id='test_customer',
            file_id='test_file',
            chunk_id='chunk_001',
            audit_level='lite',
            cost_tracking={}
        )

        # Execute validation
        result = await validation_engine.validate_single_record(test_record, context)

        # Assertions
        assert result['record_id'] == 'test_001'
        assert result['overall_status'] == 'passed'
        assert len(result['validation_results']) == 1

@pytest.mark.asyncio
async def test_rule_dependency_execution_order(self, validation_engine):
    """Test that rules execute in correct dependency order"""

    # Create rules with dependencies
    rules = [
        ValidationRule(id="rule_a", dependencies=["rule_b"], tier=ValidationTier.SIMPLE_FIE
        ValidationRule(id="rule_b", dependencies=[], tier=ValidationTier.SIMPLE_FIELD),
        ValidationRule(id="rule_c", dependencies=["rule_a", "rule_b"], tier=ValidationTier.
    ]

    rule_graph = RuleGraph.from_rules(rules)
    execution_clusters = rule_graph.get_execution_clusters(["rule_a", "rule_b", "rule_c"])

    # Assert correct execution order
    assert execution_clusters[0] == ["rule_b"]  # No dependencies
    assert execution_clusters[1] == ["rule_a"]  # Depends on rule_b
    assert execution_clusters[2] == ["rule_c"]  # Depends on rule_a and rule_b
```

```python
@pytest.mark.asyncio
async def test_cache_performance(self, validation_engine):
    """Test multi-tier caching performance"""

    cache_manager = validation_engine.cache_manager

    # Test cache miss -> S3 retrieval
    with patch.object(cache_manager.s3_client, 'get_object') as mock_s3:
        mock_s3.return_value = {'Body': MockS3Body(json.dumps({'test': 'data'}))}

        result = await cache_manager.get_rule_graph('customer_123', 'v1.0')

        # Should hit S3 and populate caches
        mock_s3.assert_called_once()
        assert 'rule_graph:customer_123:v1.0' in cache_manager.lambda_cache

    # Test cache hit
    with patch.object(cache_manager.s3_client, 'get_object') as mock_s3:
        result = await cache_manager.get_rule_graph('customer_123', 'v1.0')

        # Should not hit S3 (cached)
        mock_s3.assert_not_called()

@pytest.mark.asyncio
async def test_cost_tracking_accuracy(self, validation_engine):
    """Test cost calculation accuracy"""

    cost_tracker = CostTracker()

    job_metrics = {
        'lambda_invocations': 100,
        'lambda_gb_seconds': 250.5,
        'sqs_requests': 1500,
        's3_put_requests': 50,
        's3_get_requests': 200,
        's3_storage_gb': 5.2,
        'processing_time_hours': 0.25,
        'concurrent_jobs_on_redis': 3,
        'dynamodb_read_units': 1000,
        'dynamodb_write_units': 200,
        'records_processed': 10000
    }
```

```
        cost_breakdown = await cost_tracker.calculate_job_cost(job_metrics)

        # Verify cost calculation components
        assert cost_breakdown['total_cost'] > 0
        assert cost_breakdown['cost_per_1000_records'] == pytest.approx(
            (cost_breakdown['total_cost'] / 10) * 1000, rel=1e-3
        )
        assert 'lambda' in cost_breakdown['breakdown']
        assert 'storage' in cost_breakdown['breakdown']

class TestPerformanceBenchmarks:
    """Performance and load testing"""

    @pytest.mark.performance
    @pytest.mark.asyncio
    async def test_processing_throughput(self):
        """Test system can handle target throughput"""

        # Create test dataset
        test_records = generate_test_encounter_records(count=10000)

        start_time = time.time()

        # Process through validation engine
        validation_engine = ValidationEngine()
        await validation_engine.initialize()

        results = []
        for batch in chunk_records(test_records, batch_size=1000):
            batch_results = await validation_engine.validate_records(
                batch,
                ValidationContext(
                    customer_id='perf_test',
                    file_id='perf_test_file',
                    chunk_id=f'chunk_{len(results)}',
                    audit_level='lite',
                    cost_tracking={}
                )
            )
            results.extend(batch_results)

        end_time = time.time()
        processing_time = end_time - start_time
```

```
    # Performance assertions
    records_per_second = len(test_records) / processing_time
    assert records_per_second >= 800, f"Throughput too low: {records_per_second} records/se
    assert processing_time <= 15, f"Processing time too high: {processing_time} seconds"

@pytest.mark.performance
@pytest.mark.asyncio
async def test_memory_usage_efficiency(self):
    """Test memory usage stays within Lambda limits"""

    import psutil
    import gc

    process = psutil.Process()
    initial_memory = process.memory_info().rss / 1024 / 1024  # MB

    # Process large dataset
    large_dataset = generate_test_encounter_records(count=50000)

    validation_engine = ValidationEngine()
    await validation_engine.initialize()

    peak_memory = initial_memory

    for batch in chunk_records(large_dataset, batch_size=5000):
        await validation_engine.validate_records(batch, ValidationContext(
            customer_id='memory_test',
            file_id='memory_test_file',
            chunk_id='chunk_001',
            audit_level='lite',
            cost_tracking={}
        ))

        current_memory = process.memory_info().rss / 1024 / 1024
        peak_memory = max(peak_memory, current_memory)

        # Force garbage collection
        gc.collect()

    # Memory assertions (Lambda limit is 3008 MB)
    assert peak_memory <= 2500, f"Memory usage too high: {peak_memory} MB"

    final_memory = process.memory_info().rss / 1024 / 1024
    memory_growth = final_memory - initial_memory
```

```
        assert memory_growth <= 100, f"Memory leak detected: {memory_growth} MB growth"

class TestIntegrationScenarios:
    """End-to-end integration testing"""

    @pytest.mark.integration
    @pytest.mark.asyncio
    async def test_complete_validation_workflow(self):
        """Test complete file processing workflow"""

        # Mock AWS services
        with patch('boto3.client') as mock_boto3:
            mock_s3 = AsyncMock()
            mock_sqs = AsyncMock()
            mock_dynamodb = AsyncMock()
            mock_boto3.return_value = mock_s3

            # Setup test file
            test_file_content = generate_test_x12_file(record_count=1000)

            # Simulate S3 trigger event
            s3_event = {
                'Records': [{
                    's3': {
                        'bucket': {'name': 'test-bucket'},
                        'object': {'key': 'test-customer/test-file.x12'}
                    }
                }]
            }

            # Process through file processor
            from src.file_processor import handler as file_processor_handler

            mock_s3.get_object.return_value = {
                'Body': MockS3Body(test_file_content),
                'ContentLength': len(test_file_content)
            }

            result = await file_processor_handler(s3_event, {})

            # Verify SQS messages sent
            assert mock_sqs.send_message.called

            # Simulate validation processing
```

```
        from src.validation_engine import handler as validation_handler

        sqs_event = {
            'Records': [{
                'body': json.dumps({
                    'bucket': 'test-bucket',
                    'key': 'test-customer/processed/chunk_001.json',
                    'customer_id': 'test-customer',
                    'processing_strategy': {
                        'audit_level': 'lite',
                        'chunk_size': 1000
                    }
                })
            }]
        }

        validation_result = await validation_handler(sqs_event, {})

        # Verify results stored
        assert mock_s3.put_object.called

        # Verify audit trail created
        put_object_calls = mock_s3.put_object.call_args_list
        audit_calls = [call for call in put_object_calls if 'audit' in str(call)]
        assert len(audit_calls) > 0

@pytest.mark.integration
@pytest.mark.asyncio
async def test_error_handling_and_recovery(self):
    """Test system handles errors gracefully"""

    validation_engine = ValidationEngine()

    # Test invalid input handling
    invalid_record = {'invalid': 'data', 'missing_required_fields': True}

    result = await validation_engine.validate_single_record(
        invalid_record,
        ValidationContext(
            customer_id='error_test',
            file_id='error_test_file',
            chunk_id='chunk_001',
            audit_level='verbose',
            cost_tracking={}
```

```
            )
        )

        # Should handle gracefully
        assert result['overall_status'] == 'error'
        assert len(result['validation_results']) >= 0  # May have some basic validations

        # Test network failure recovery
        with patch.object(validation_engine.cache_manager, 'get_rule_graph') as mock_cache:
            mock_cache.side_effect = Exception("Network timeout")

            # Should fall back to basic validation
            with pytest.raises(Exception):
                await validation_engine.validate_single_record(
                    {'id': 'test'},
                    ValidationContext('test', 'test', 'test', 'lite', {})
                )

class TestSecurityAndCompliance:
    """Security and HIPAA compliance testing"""

    @pytest.mark.security
    def test_phi_detection_and_scrubbing(self):
        """Test PHI detection in logs and outputs"""

        data_protection = DataProtectionService()

        # Test content with PHI
        phi_content = """
        Patient John Doe, SSN: 123-45-6789,
        Phone: 5551234567,
        Email: john.doe@email.com
        """

        scan_result = asyncio.run(data_protection.scan_for_phi(phi_content))

        assert scan_result['phi_detected'] == True
        assert len(scan_result['findings']) >= 3  # SSN, phone, email

        # Test audit data anonymization
        audit_data = {
            'customer_id': 'test_customer',
            'ip_address': '192.168.1.100',
            'user_agent': 'Mozilla/5.0...',
```

```
            'processing_details': 'Processed encounter data for John Doe'
        }

        anonymized = asyncio.run(data_protection.anonymize_audit_data(audit_data))

        assert 'ip_address' not in anonymized
        assert 'ip_address_hash' in anonymized
        assert len(anonymized['ip_address_hash']) <= 16  # Truncated hash

    @pytest.mark.security
    @pytest.mark.asyncio
    async def test_encryption_and_access_control(self):
        """Test data encryption and access controls"""

        # Test S3 encryption
        with patch('boto3.client') as mock_boto3:
            mock_s3 = AsyncMock()
            mock_boto3.return_value = mock_s3

            audit_logger = AuditLogger()

            test_audit_data = {
                'customer_id': 'test_customer',
                'resource': 'patient_data',
                'action': 'validation',
                'contains_phi': True,
                'request_id': 'req_12345'
            }

            await audit_logger.log_data_access(test_audit_data)

            # Verify S3 put_object called with encryption
            mock_s3.put_object.assert_called()
            call_args = mock_s3.put_object.call_args
            assert 'ServerSideEncryption' in call_args.kwargs
            assert call_args.kwargs['ServerSideEncryption'] == 'aws:kms'

    @pytest.mark.compliance
    def test_audit_trail_completeness(self):
        """Test audit trails meet compliance requirements"""

        # Test audit event structure
        audit_event = {
            'timestamp': '2025-05-24T10:30:00Z',
```

```
            'event_type': 'data_access',
            'customer_id': 'test_customer',
            'user_id': 'user_123',
            'resource_accessed': 'encounter_data',
            'action': 'validate',
            'ip_address_hash': 'abc123def456',
            'request_id': 'req_789',
            'compliance_flags': {
                'contains_phi': True,
                'minimum_necessary': True,
                'authorized_purpose': 'quality_improvement'
            }
        }

        # Verify required fields present
        required_fields = [
            'timestamp', 'event_type', 'customer_id', 'resource_accessed',
            'action', 'request_id', 'compliance_flags'
        ]

        for field in required_fields:
            assert field in audit_event, f"Required audit field '{field}' missing"

        # Verify compliance flags
        compliance = audit_event['compliance_flags']
        assert isinstance(compliance['contains_phi'], bool)
        assert isinstance(compliance['minimum_necessary'], bool)
        assert compliance['authorized_purpose'] in [
            'quality_improvement', 'fraud_detection', 'regulatory_compliance'
        ]

# Test Utilities
class MockS3Body:
    def __init__(self, content):
        self.content = content.encode() if isinstance(content, str) else content

    def read(self):
        return self.content

def generate_test_encounter_records(count: int) -> List[Dict]:
    """Generate test encounter records for testing"""

    records = []
    for i in range(count):
```

```
        record = {
            'id': f'encounter_{i:06d}',
            'patient_id': f'patient_{i % 1000:04d}',
            'diagnosis_codes': ['Z23.1', 'M25.50', 'I10'],
            'procedure_codes': ['99213', '90715'],
            'service_date': '2025-05-24',
            'provider_npi': '1234567890',
            'place_of_service': '11',
            'claim_amount': round(random.uniform(50.0, 500.0), 2)
        }
        records.append(record)
    return records

def generate_test_x12_file(record_count: int) -> str:
    """Generate test X12 EDI file content"""

    x12_content = "ISA*00*          *00*          *ZZ*SENDER_ID      *ZZ*RECEIVER_ID    *250524

    for i in range(record_count):
        x12_content += f"CLM*{i:06d}*100.00*1*1*1:B:1*Y*A*Y*Y~\n"
        x12_content += f"HI*BK:Z231*BF:M2550*BF:I10~\n"

    x12_content += "IEA*1*000000001~\n"

    return x12_content

def chunk_records(records: List[Dict], batch_size: int) -> List[List[Dict]]:
    """Chunk records into batches for processing"""

    for i in range(0, len(records), batch_size):
        yield records[i:i + batch_size]
```

Error Handling & Recovery

Comprehensive Error Management

python

```python
class ErrorHandler:
    def __init__(self):
        self.error_categories = {
            'validation_error': {'severity': 'low', 'retry': True, 'max_retries': 3},
            'data_format_error': {'severity': 'medium', 'retry': False, 'alert': True},
            'system_error': {'severity': 'high', 'retry': True, 'max_retries': 2},
            'cost_limit_exceeded': {'severity': 'medium', 'retry': False, 'alert': True},
            'rate_limit_exceeded': {'severity': 'low', 'retry': True, 'backoff': True}
        }

    async def handle_error(self, error: Exception, context: Dict) -> Dict:
        """Centralized error handling with categorization and recovery"""

        error_type = self.categorize_error(error)
        error_config = self.error_categories.get(error_type, {})

        error_response = {
            'error_id': str(uuid.uuid4()),
            'error_type': error_type,
            'error_message': str(error),
            'context': context,
            'timestamp': datetime.utcnow().isoformat(),
            'recovery_action': None,
            'customer_impact': self.assess_customer_impact(error_type, context)
        }

        # Execute recovery actions
        if error_config.get('retry', False):
            recovery_result = await self.attempt_recovery(error, context, error_config)
            error_response['recovery_action'] = recovery_result

        # Send alerts if required
        if error_config.get('alert', False):
            await self.send_error_alert(error_response)

        # Log error for analysis
        await self.log_error(error_response)

        return error_response

    def categorize_error(self, error: Exception) -> str:
        """Categorize errors for appropriate handling"""
```

```
        error_str = str(error).lower()

        if 'validation' in error_str or 'rule' in error_str:
            return 'validation_error'
        elif 'format' in error_str or 'parse' in error_str:
            return 'data_format_error'
        elif 'cost' in error_str or 'budget' in error_str:
            return 'cost_limit_exceeded'
        elif 'rate limit' in error_str or 'throttle' in error_str:
            return 'rate_limit_exceeded'
        elif isinstance(error, (ConnectionError, TimeoutError)):
            return 'system_error'
        else:
            return 'unknown_error'

    async def attempt_recovery(self, error: Exception, context: Dict, config: Dict) -> Dict:
        """Attempt automated error recovery"""

        recovery_actions = []

        # Implement exponential backoff for retryable errors
        if config.get('backoff', False):
            retry_count = context.get('retry_count', 0)
            backoff_time = min(2 ** retry_count, 300)  # Max 5 minutes

            await asyncio.sleep(backoff_time)
            recovery_actions.append(f"Applied exponential backoff: {backoff_time}s")

        # Clear cache if system error
        if 'system_error' in str(error):
            await self.clear_stale_cache(context.get('customer_id'))
            recovery_actions.append("Cleared potentially stale cache")

        # Switch to degraded mode if needed
        if context.get('retry_count', 0) >= 2:
            recovery_actions.append("Switched to degraded processing mode")

        return {
            'attempted': True,
            'actions': recovery_actions,
            'timestamp': datetime.utcnow().isoformat()
        }

    async def send_error_alert(self, error_response: Dict):
```

```
        """Send error alerts to appropriate channels"""

        severity = self.error_categories.get(
            error_response['error_type'], {}
        ).get('severity', 'medium')

        alert_data = {
            'alert_type': 'system_error',
            'severity': severity,
            'error_summary': error_response['error_message'][:200],
            'customer_id': error_response['context'].get('customer_id'),
            'error_id': error_response['error_id'],
            'impact_assessment': error_response['customer_impact']
        }

        # High severity errors -> immediate Slack/PagerDuty
        if severity == 'high':
            await self.send_critical_alert(alert_data)

        # Medium severity -> Slack notification
        elif severity == 'medium':
            await self.send_slack_notification(alert_data)

        # All errors -> CloudWatch alarm
        await self.trigger_cloudwatch_alarm(alert_data)
```

Circuit Breaker Pattern

python

```python
class CircuitBreaker:
    def __init__(self, failure_threshold: int = 5, recovery_timeout: int = 60):
        self.failure_threshold = failure_threshold
        self.recovery_timeout = recovery_timeout
        self.failure_count = 0
        self.last_failure_time = None
        self.state = 'CLOSED'  # CLOSED, OPEN, HALF_OPEN

    async def call(self, func, *args, **kwargs):
        """Execute function with circuit breaker protection"""

        if self.state == 'OPEN':
            if self._should_attempt_reset():
                self.state = 'HALF_OPEN'
            else:
                raise CircuitBreakerOpenError("Circuit breaker is OPEN")

        try:
            result = await func(*args, **kwargs)
            self._on_success()
            return result

        except Exception as e:
            self._on_failure()
            raise

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

        if self.last_failure_time is None:
            return True

        return (time.time() - self.last_failure_time) >= self.recovery_timeout

    def _on_success(self):
        """Handle successful call"""

        self.failure_count = 0
        self.state = 'CLOSED'

    def _on_failure(self):
        """Handle failed call"""
```

```
            self.failure_count += 1
            self.last_failure_time = time.time()

            if self.failure_count >= self.failure_threshold:
                self.state = 'OPEN'

    class CircuitBreakerOpenError(Exception):
        """Raised when circuit breaker is open"""
        pass

    # Usage in validation engine
    class ValidationEngineWithCircuitBreaker(ValidationEngine):
        def __init__(self):
            super().__init__()
            self.redis_circuit_breaker = CircuitBreaker(failure_threshold=3, recovery_timeout=30)
            self.s3_circuit_breaker = CircuitBreaker(failure_threshold=5, recovery_timeout=60)

        async def load_rule_graph_with_protection(self, customer_id: str):
            """Load rule graph with circuit breaker protection"""

            try:
                # Try Redis with circuit breaker
                return await self.redis_circuit_breaker.call(
                    self.cache_manager.get_rule_graph_from_redis, customer_id
                )
            except CircuitBreakerOpenError:
                # Fall back to S3 with its own circuit breaker
                return await self.s3_circuit_breaker.call(
                    self.cache_manager.get_rule_graph_from_s3, customer_id
                )
```

Conclusion

This High-Level Design document provides a comprehensive technical blueprint for implementing the Healthcare Data Validation Rule Engine. The design addresses real-world constraints while maintaining the core innovation of cost-effective, scalable healthcare data validation.

Key Technical Achievements

1. Production-Ready Architecture

- Multi-tier caching strategy eliminates performance bottlenecks

- Adaptive batch processing optimizes cost and performance
- Comprehensive error handling and recovery mechanisms
- Circuit breaker patterns prevent cascade failures

2. Cost-Effective Serverless Design

- Achieves 99%+ cost reduction through intelligent resource utilization
- Dynamic scaling prevents over-provisioning
- Real-time cost monitoring and budget controls
- Transparent pricing model with predictable costs

3. Healthcare Compliance Focus

- HIPAA-compliant data handling and audit trails
- Automated PHI detection and anonymization
- 7-year audit retention with intelligent lifecycle management
- Comprehensive access logging and monitoring

4. Scalable Performance Architecture

- Handles 800+ records/second processing throughput
- Sub-10 second response times with intelligent caching
- Auto-scaling from 1K to 10M+ records seamlessly
- Memory-efficient design within Lambda constraints

Implementation Readiness

Technical Specifications:

- Complete component specifications with AWS service configurations
- Detailed API specifications with authentication and rate limiting
- Comprehensive data models and schema definitions
- Production-ready CI/CD pipeline with automated testing

Quality Assurance:

- Unit testing framework with 90%+ code coverage targets
- Integration testing for end-to-end workflow validation
- Performance benchmarking with specific throughput requirements

- Security testing for HIPAA compliance verification

Operational Excellence:

- CloudWatch monitoring with custom healthcare metrics
- Distributed tracing for performance optimization
- Cost tracking and optimization recommendations
- Comprehensive error handling and alerting

Next Steps for Implementation

Phase 1 (Months 1-3): Core Development

- Implement validation engine with extensible architecture
- Build multi-tier caching with Redis and S3 integration
- Create file processing pipeline with SQS orchestration
- Develop basic audit and compliance features

Phase 2 (Months 4-6): Production Hardening

- Implement comprehensive error handling and circuit breakers
- Add cost monitoring and budget controls
- Build customer dashboard and self-service features
- Complete security and HIPAA compliance implementation

Phase 3 (Months 7-9): Market Launch

- Deploy production infrastructure with Terraform
- Launch customer onboarding and payment processing
- Implement advanced monitoring and optimization
- Begin customer acquisition and support operations

This HLD provides engineering teams with everything needed to build a production-ready healthcare data validation system that democratizes access to enterprise-grade validation capabilities through innovative technology and business model design.

Document Version: 1.0
Last Updated: May 24, 2025

Classification: Technical Implementation - Engineering Ready
Review Status: Ready for Development Team Review