Azure Logic Apps Actions: Comprehensive Guide for Health Insurance

Table of Contents

Introduction

Logic Apps provides a rich set of actions for workflow automation. Each action serves a specific purpose in the workflow:

- **Data manipulation**: ParseJson, Compose, SetVariable
- **Conditional logic**: If, Switch, Scope
- **External integrations**: Http, ApiConnection
- **Data storage**: SQL, CosmosDB, Service Bus
- **Notifications**: Email, Teams, Event Grid
- **Control flow**: Foreach, Until, Do-Until

The selection of actions depends on the business requirement, data flow, error handling needs, and system constraints.

Project 1: Claims Processing - Action Breakdown

Workflow Overview

Trigger (Service Bus) → Parse Data → Validate → Check Eligibility → Fraud Detection → Calculate Benefits → Route Decision → Database Update → Notifications → Complete

1.1 InitializeVariable Action

Purpose: Set up baseline variables for claim processing workflow

Why Used:

- Claim processing involves multiple decision points and calculations
- Variables track state throughout the workflow (claim status, amounts, fraud scores)
- Enables easy reference and updates across multiple actions
- Provides a single source of truth for claim metadata

Example Usage:

```
{
"claimId": "CLM-2024-001",
"claimStatus": "Processing",
"approvalAmount": 0,
"fraudScore": 0,
"requiresManualReview": false,
"processingNotes": []
}
```

Key Variables:

- claimId: Unique identifier for the entire workflow
- claimStatus: Tracks state (Processing → Approved/Denied/Pending Review)
- approvalAmount: Calculated benefit amount (starts at 0, updated after calculation)
- fraudScore: ML output (0-1 scale) for risk assessment
- requiresManualReview: Boolean flag to route to human review

When to Use:

- Complex multi-step workflows with interdependent actions
- When you need to pass data between disconnected actions
- For audit trail and logging purposes
- To avoid repetitive data extraction

1.2 ParseJson Action

Purpose: Convert unstructured claim message into typed JSON object

Why Used:

- Service Bus receives claims in various formats (XML, CSV, unstructured JSON)
- ParseJson validates structure and provides type safety
- Enables intellisense and type checking in subsequent actions
- Fails fast if data format is invalid, preventing downstream errors

Example Input:

```
{
"claimId": "CLM-2024-001",
"memberId": "MEM-123456",
"claimAmount": 5000,
"diagnosisCodes": ["E11", "I10"],
"procedureCodes": ["99213", "80053"]
}
```

Schema Definition:

```
{
"type": "object",
"properties": {
"claimId": { "type": "string" },
"memberId": { "type": "string" },
"claimAmount": { "type": "number" },
"diagnosisCodes": { "type": "array", "items": { "type": "string" } }
},
"required": ["claimId", "memberId"]
}
```

Benefits:

- **Type Safety**: Prevents operations on wrong data types
- **Validation**: Schema enforcement catches malformed claims early
- **Error Handling**: ParseJson failures route to error handling scope
- **Documentation**: Schema serves as contract specification

When to Use:

- First action after trigger to normalize input
- When data comes from external APIs or message queues
- Before any conditional logic that depends on specific fields
- To transform trigger body to usable format

1.3 SetVariable Action

Purpose: Update tracked variables based on processing outcomes

Why Used:

- Claims processing requires multiple decision points
- Each decision updates the claim status and associated metadata
- SetVariable provides clean way to update state
- Enables conditional routing based on accumulated state

Example Usages:

```
// After eligibility check
{
"name": "claimStatus",
"value": "Approved"
}

// After fraud detection
{
"name": "fraudScore",
"value": "@body('Run_Fraud_Detection_Model')?['fraudScore']"
}

// Building processing notes
{
"name": "processingNotes",
"value": "@concat(variables('processingNotes'), array('Eligibility verified'))"
}
```

Why Better Than:

- **Compose**: Compose is for one-time data transformation, not state updates
- **Inline Variables**: Cannot be referenced in expressions without Compose

- **SQL Direct**: SetVariable is in-memory and fast; database writes are slower

When to Use:

- After successful validation/check
- To accumulate results from multiple sources
- Before conditional logic that depends on the value
- To build audit trails in array format

1.4 If (Conditional) Action

Purpose: Create branching logic based on validation criteria

Why Used:

- Claims have multiple acceptance criteria (eligibility, coverage, fraud score)
- Different paths for approved vs denied claims
- Prevents processing invalid claims further downstream
- Enables early rejection to save processing costs

Example Logic:

```
{
"and": [
{
"not": {
"equals": ["@body('Parse_Claims_Message')?['memberId']", ""]
}
},
{
"not": {
"equals": ["@body('Parse_Claims_Message')?['claimId']", ""]
}
},
{
"greater": ["@body('Parse_Claims_Message')?['claimAmount']", 0]
}
]
}
```

Decision Criteria:

1. **Member ID present**: Claim cannot be processed without member
2. **Claim ID present**: Unique identification required for tracking
3. **Amount positive**: $0 claims are invalid and create reconciliation issues

True Path Actions:

- Eligibility verification
- Fraud detection
- Benefit calculation

False Path Actions:

- Mark as "Invalid Data"
- Log error details
- Send rejection notification

When to Use:

- At validation gates (first action after parsing)
- Before expensive operations (API calls, ML models)
- To prevent cascading errors
- To separate happy path from error handling

1.5 Http Action (Call External API)

Purpose: Check member eligibility against external eligibility service

Why Used:

- Eligibility data lives in separate system (member master, coverage database)
- Real-time verification ensures current status
- External system is source of truth
- Logic Apps cannot query external databases directly without Http

Example Configuration:

```
{
"type": "Http",
"inputs": {
"method": "POST",
"uri": "https://eligibility-api.company.com/verify-eligibility",
"headers": {
"Content-Type": "application/json",
"Authorization": "Bearer @{variables('bearerToken')}"
},
"body": {
"memberId": "@body('Parse_Claims_Message')?['memberId']",
"serviceDate": "@body('Parse_Claims_Message')?['serviceDate']",
"procedureCodes": "@body('Parse_Claims_Message')?['procedureCodes']"
}
},
"retryPolicy": {
"type": "exponential",
"count": 3,
"interval": "PT5S",
"maximumInterval": "PT30S"
}
}
```

Why Retry Policy Matters:

- External services may be temporarily unavailable
- Network timeouts are common in microservices
- Exponential backoff prevents overwhelming failing service

- Retry count = 3 balances reliability with timeout (max ~45 seconds)

When to Use:

- When data doesn't exist in current system
- For real-time verification (eligibility changes frequently)
- To avoid data duplication
- When external system is authoritative

Alternatives Not Used:

- **SQL Direct**: Only works for data in SQL Server
- **CosmosDB Direct**: Cannot query other companies' CosmosDB
- **Compose**: Cannot make external calls

1.6 ApiConnection Action (SQL Database)

Purpose: Lookup member benefit details from SQL database

Why Used:

- Benefit structure (copay, coinsurance, deductible) stored in relational database
- SQL query performance is optimal for structured data
- Member-specific benefit data requires WHERE clause filtering
- ApiConnection abstracts SQL connection management

Example Query:

```
SELECT CoverageTier, Copay, Coinsurance, DeductibleMet
FROM MemberCoverage
WHERE MemberId = '@{body('Parse_Claims_Message')?['memberId']}'
```

Data Retrieved:

- **CoverageTier**: Bronze/Silver/Gold (affects coinsurance percentage)
- **Copay**: Fixed member cost per visit
- **Coinsurance**: Percentage member pays after deductible
- **DeductibleMet**: Whether annual deductible already satisfied

Why ApiConnection Instead of Http:

- ApiConnection has built-in SQL Server connector
- Handles authentication automatically
- Query builder available in UI
- Native parameter binding prevents SQL injection

When to Use:

- Lookup operations (SELECT statements)
- When data is in SQL Server
- For high-frequency queries (cached connections)
- When transactional consistency needed

1.7 Azure ML Http Action (Fraud Detection)

Purpose: Call machine learning model to score claim fraud probability

Why Used:

- Manual fraud detection is time-consuming and inconsistent
- ML models identify patterns humans might miss
- Fraud costs insurers millions annually
- Automated scoring enables real-time claim decisions

Example Input:

```
{
"claimAmount": 5000,
"memberAge": 35,
"claimFrequency": 2, // claims in last 90 days
"providerRiskScore": 0.3,
"diagnosisCodes": ["E11", "I10"],
"procedureCodes": ["99213", "80053"]
}
```

Model Output:

```
{
"fraudScore": 0.15, // 0-1 scale, >0.7 = high risk
"confidence": 0.92,
"riskFactors": ["unusual-claim-amount", "high-provider-frequency"]
}
```

Scoring Interpretation:

- **0.0-0.3**: Low fraud risk → Auto-approve likely
- **0.3-0.7**: Medium risk → May require additional review
- **0.7-1.0**: High fraud risk → Manual review required

Why ML Instead of Rules:

- Rules-based systems (IF amount > $10k) have false positives
- ML models learn patterns from historical data
- Adapts as fraud patterns evolve
- Consider context (provider history, member demographics)

When to Use:

- When pattern detection is needed
- For high-value decisions (fraud detection, credit scoring)
- When rules change frequently
- To reduce manual review workload

1.8 Scope Action (Benefit Calculation)

Purpose: Encapsulate benefit calculation logic with error handling

Why Used:

- Complex calculation requires multiple steps
- Scope action provides logical grouping
- Failures in calculation don't fail entire workflow
- Isolates calculation errors for specific handling

Contained Actions:

```
{
"Calculate_Benefit": {
"type": "Compose",
"inputs": {
"claimAmount": 5000,
"copay": 25,
"coinsurance": 0.20,
"deductibleMet": true,
"approvalPercentage": 0.80
}
},
"Set_Approval_Amount": {
"type": "SetVariable",
"inputs": {
"name": "approvalAmount",
"value": "@{mul(body('Parse_Claims_Message')?['claimAmount'], 0.8)}"
}
}
}
```

Benefit Calculation Formula:
Approval Amount = Claim Amount × Coverage Percentage

- Copay × Frequency
- (Claim Amount × Coinsurance) if deductible not met

Example: $5,000 claim with 80% coverage, $25 copay, 20% coinsurance
= ($5,000 × 0.80) - $25 = $3,975

Why Scope Instead of Inline:

- Error in calculation doesn't stop entire workflow
- Clear visual grouping in Logic App designer
- Can add specific error handling for calculation failures
- Easier to test and debug complex logic

When to Use:

- Multi-step calculations
- When you want isolated error handling

- To improve workflow readability
- For business logic that may change

1.9 Foreach Loop (Not in Project 1, but common pattern)

Purpose: Process multiple items in array sequentially or in parallel

Why Used:

- Claims may have multiple procedure codes
- Each procedure might have different coverage rules
- Sequential processing ensures order preservation
- Parallel processing improves performance

Example:
```
{
"type": "Foreach",
"foreach": "@body('Parse_Claims_Message')?['procedureCodes']",
"actions": {
"Check_Procedure_Coverage": {
"type": "If",
"expression": {
"contains": ["@variables('coveredProcedures')]", "@item()"
],
"actions": { /* approve */ }
}
}
}
```

When to Use:

- Array processing
- Multiple items need same logic
- Order matters (sequential)
- Performance critical (parallel)

1.10 ApiConnection Action (Update Database)

Purpose: Insert claim processing result into database for audit trail

Why Used:

- All claim decisions must be logged for compliance
- Creates immutable audit trail
- Enables future analysis (appeals, pattern detection)
- Required for HIPAA/regulatory compliance

Data Recorded:

```
{
"claimId": "CLM-2024-001",
"memberId": "MEM-123456",
"claimAmount": 5000,
"approvalAmount": 4000,
"claimStatus": "Approved",
"fraudScore": 0.15,
"requiresManualReview": false,
"processedDate": "2024-12-25T07:05:00Z",
"processingNotes": ["Eligibility verified", "Fraud score acceptable"]
}
```

Why Database Insert:

- Permanent record for audit
- Enables querying historical claims
- Source of truth for claims status
- Supports analytics and reporting

When to Use:

- After major workflow steps
- When compliance audit trail required
- For operational metrics
- Before final notification

1.11 Email Actions (Notifications)

Purpose: Notify provider and member of claim decision

Why Used:

- Stakeholders need immediate notification
- Email provides persistent notification (not real-time)
- Can include claim details and next steps
- Creates communication record

Provider Email:
To: provider@hospital.com
Subject: Claim Processed - Claim ID: CLM-2024-001
Body:
Claim Amount: $5,000
Approved Amount: $4,000
Status: Approved
Next: Settlement processing within 3 business days

Member Email:
To: member@personal.com
Subject: Your Claim Has Been Processed
Body:
Your claim CLM-2024-001 has been processed.
Approved Amount: $4,000
Status: Approved
Your member responsibility: $1,000
Claim payment expected: 5-7 business days

Why Separate Emails:

- Providers need payment/settlement info
- Members need cost-sharing info
- Different templates for different audiences
- Compliance: Different data for different parties

When to Use:

- After final decision (approval/denial)
- For important stakeholder notifications
- When audit trail needed
- For user-facing status updates

1.12 Complete Queue Message Action

Purpose: Remove successfully processed message from Service Bus queue

Why Used:

- Service Bus keeps messages until explicitly deleted
- Prevents re-processing same claim multiple times
- Signals successful workflow completion
- Deadletter queue holds failed messages for investigation

Why Not Auto-Delete:

- Message stays in queue if workflow fails
- Allows reprocessing on Logic App restart
- Dead-letter queue captures permanently failed messages
- Manual retry possible for transient failures

When to Use:

- At end of successful workflow path
- Before returning response
- After all side effects (database, notifications) complete

1.13 Scope (Error Handling)

Purpose: Catch and handle errors from processing actions

Why Used:

- Prevents workflow failure from crashing entire pipeline
- Logs errors for investigation
- Moves failed message to dead-letter queue
- Sends alert to operations team

Error Actions:

```
{
"Log_Error_Details": {
"type": "Http",
"inputs": {
```

```
"uri": "https://ApplicationInsights-endpoint/traces",
"body": {
"message": "Claim processing failed",
"claimId": "@variables('claimId')",
"error": "@result('Validate_Claim_Data')"
}
}
},
"Move_To_Dead_Letter": {
"type": "ApiConnection",
"inputs": {
"path": "/messages/deadletter",
"body": "@triggerBody()"
}
}
}
```

When to Use:

- After risky operations (external API calls)
- Around third-party integrations
- Before marking as complete
- For compliance/audit requirements

Project 2: Real-time Policy Eligibility Verification - Action Breakdown

Workflow Overview

Http Request Trigger → Initialize Variables → Query CosmosDB
→ Check Coverage Rules → Calculate Deductible → Evaluate Pre-Auth
→ Compile Response → Http Response

2.1 Http Request Trigger

Purpose: Accept eligibility check requests from provider systems

Why Used:

- Providers need real-time eligibility during patient check-in
- Http Request triggers enable synchronous workflows
- Request schema validation ensures data quality
- Immediate response improves provider experience

Request Schema:
```
{
"type": "object",
"properties": {
"memberId": { "type": "string" },
"serviceDate": { "type": "string", "format": "date" },
"procedureCodes": { "type": "array", "items": { "type": "string" } },
"facilityType": { "type": "string" },
"providerId": { "type": "string" }
},
"required": ["memberId", "serviceDate", "procedureCodes"]
}
```

Why Required Fields:

- **memberId**: Cannot check eligibility without member
- **serviceDate**: Coverage varies by date (effective/termination dates)
- **procedureCodes**: Some procedures have restrictions

When to Use:

- API-style workflows requiring synchronous response
- When immediate reply needed
- For external system integration
- When caller expects 200 OK response

2.2 ApiConnection (CosmosDB Query)

Purpose: Retrieve member policy document in real-time

Why Used:

- Policies change frequently (enrollments, terminations)
- CosmosDB provides <10ms latency for lookups
- JSON documents store nested policy data perfectly
- Global distribution supports multi-region providers

Query:
```
SELECT p.memberId, p.policyNumber, p.benefits, p.exclusions
FROM Policies p
WHERE p.memberId = @memberId
AND p.effectiveDate <= @serviceDate
AND p.terminationDate > @serviceDate
```

Why CosmosDB Instead of SQL Server:

- **JSON Native**: Policies are hierarchical (benefits → copay, coinsurance, limits)
- **Speed**: <10ms vs 50-100ms for SQL
- **Global Scale**: Replica in each region for local reads
- **Flexible Schema**: New benefit types don't require schema changes

Policy Document Structure:
```
{
"memberId": "MEM-123456",
"policyNumber": "POL-2024-001",
"effectiveDate": "2024-01-01",
"terminationDate": "2024-12-31",
"benefits": {
"inNetwork": {
"copay": 25,
"coinsurance": 0.20,
"deductible": 1500
},
"outOfNetwork": {
"copay": 50,
"coinsurance": 0.40,
"deductible": 3000
}
},
"exclusions": {
"excludedProcedures": ["90837", "90834"],
"excludedDiagnoses": ["Z00.00"] // preventive exams
}
}
```

When to Use:

- Fast lookups needed
- Hierarchical data structures
- Multi-region performance critical
- Document-oriented data model

2.3 If (Check Policy Found)

Purpose: Branch logic based on whether active policy exists

Why Used:

- Not all members have active policies (terminated, never enrolled)
- Different response needed if no policy found
- Prevents null reference errors in subsequent actions
- Enables proper error messaging to provider

True Path (Policy Found):

- Proceed with coverage rule evaluation
- Calculate benefits
- Check pre-auth requirements

False Path (No Policy):

- Return "No active policy" message
- Prevent claims from being processed
- Alert provider to verify enrollment status

When to Use:

- After lookup operations
- When null/missing data possible
- Before operations assuming data exists
- For user-facing responses

2.4 Foreach (Evaluate Procedures)

Purpose: Check each requested procedure against exclusions list

Why Used:

- Claims may include multiple procedures
- Each procedure has different coverage rules
- Some procedures excluded entirely (experimental, non-approved)
- Must check all procedures before approving claim

Example:
```
{
"procedureCodes": ["99213", "80053", "90837"],
"excludedProcedures": ["90837"] // psychotherapy
}
```

Evaluation Logic:

- Loop through each procedure code
- Check if in policy exclusions
- If found: set `isEligible = false`
- If not found: continue checking

When to Use:

- Array/list processing
- When each item needs evaluation
- Multiple items affect single decision
- Sequential processing important

2.5 ApiConnection (SQL - Deductible Calculation)

Purpose: Calculate remaining deductible from year-to-date claims

Why Used:

- Deductible status affects member cost-sharing
- Claims database has authoritative record of paid amounts
- Real-time calculation ensures accurate cost-sharing
- SQL query efficiently aggregates claims

Query:
```
SELECT
ISNULL(SUM(claimAmount), 0) as YearToDateClaims,
@deductible - ISNULL(SUM(claimAmount), 0) as DeductibleRemaining
FROM Claims
WHERE memberId = '@{memberId}'
AND YEAR(serviceDate) = YEAR('@{serviceDate}')
AND claimStatus IN ('Approved', 'Paid')
```

Calculation Example:

- Member deductible: $1,500

- YTD approved claims: $800
- Deductible remaining: $700

Why Not in CosmosDB:

- Deductible calculated from claims (different document)
- Requires aggregation across many claims
- SQL is superior for aggregation queries
- Claims in SQL Server, policies in CosmosDB (separated concerns)

When to Use:

- Aggregation queries (SUM, COUNT, AVG)
- When joining multiple tables
- Transactional consistency needed
- Calculation depends on historical data

2.6 Http Response Action

Purpose: Return eligibility determination to calling provider system

Why Used:

- Http Request trigger requires Http Response
- Synchronous workflows need immediate feedback
- Response format expected by provider integration
- Status codes indicate success/failure to calling system

Success Response (200 OK):
```
{
"isEligible": true,
"benefits": {
"copay": 25,
"coinsurance": 0.20,
"deductibleRemaining": 700,
"outOfPocketMax": 5000
},
"preAuthRequired": false,
"validUntil": "2024-12-26T07:05:00Z"
}
```

Error Response (400 Bad Request):
```
{
"isEligible": false,
"eligibilityReason": "No active policy found for member",
"validUntil": "2024-12-25T12:05:00Z"
}
```

Response Properties:

- **statusCode**: 200 (success), 400 (bad request), 500 (server error)
- **body**: JSON response matching provider expectations
- **headers**: Content-Type, custom headers if needed

When to Use:

- Http Request trigger workflows (must respond)
- When immediate feedback required
- For API-style integrations
- Synchronous call-response pattern

2.7 Caching Strategy (Best Practice)

Purpose: Cache eligibility checks to improve response time

Why Used:

- Same patient checks multiple times during visit
- Database queries add latency
- Cache reduces load on backend systems
- Eligibility valid for 24 hours (changes rare)

Implementation Pattern:
```
{
"cacheKey": "eligibility-MEM-123456-2024-12-25",
"ttl": 86400, // 24 hours
"pattern": "eligibility-{memberId}-{serviceDate}"
}
```

Cache Miss Path:

1. Check cache (Redis/Azure Cache for Redis)
2. If miss: query CosmosDB + SQL
3. Store result in cache
4. Return response

Cache Hit Path:

1. Check cache
2. If hit: return cached response
3. No database queries needed
4. Response in <5ms

When to Use:

- Frequently accessed data
- Data that doesn't change often
- Performance critical operations
- Read-heavy workloads

Project 3: Member Enrollment and Onboarding - Action Breakdown

Workflow Overview

Http Request Trigger → Validate Data → Create CRM Contact
→ Generate Policy → Create Dependents → Generate ID Card
→ Create Portal Account → Send Emails → Log Completion

3.1 Validate Enrollment Data (If Action)

Purpose: Ensure all required enrollment fields present before processing

Why Used:

- Incomplete enrollments create orphaned records in multiple systems
- CRM creation fails if required fields missing
- Better to validate upfront than handle failures downstream
- Prevents inconsistent data in database

Required Fields:
```
{
"firstName": "Required - CRM contact creation",
"lastName": "Required - CRM contact creation",
"dateOfBirth": "Required - ID card and claims routing",
"email": "Required - portal account and communications",
"policyType": "Required - benefits determination"
}
```

Validation Expression:
```
{
"and": [
{ "not": { "equals": ["@triggerBody()?['firstName']", ""] } },
{ "not": { "equals": ["@triggerBody()?['lastName']", ""] } },
{ "not": { "equals": ["@triggerBody()?['dateOfBirth']", ""] } },
{ "not": { "equals": ["@triggerBody()?['email']", ""] } },
{ "not": { "equals": ["@triggerBody()?['policyType']", ""] } }
]
}
```

When Validation Fails:

- Return 400 Bad Request immediately
- List missing required fields
- No CRM/database updates occur
- Member can retry with correct data

When to Use:

- Before resource creation
- Validate required vs optional fields
- Prevent cascading failures

- Fail fast principle

3.2 Http Action (Create Member in CRM)

Purpose: Create member contact record in Dynamics 365 CRM

Why Used:

- CRM is master system for member relationships
- Enables future marketing, support interactions
- Creates single source of truth for member data
- Required before policy creation

CRM Contact Fields:
```
{
"firstname": "John",
"lastname": "Doe",
"birthdate": "1990-01-15",
"emailaddress1": "john.doe@email.com",
"telephone1": "555-0123",
"address1_line1": "123 Main St",
"address1_city": "New York",
"address1_stateorprovince": "NY",
"address1_postalcode": "10001",
"new_enrollmentsource": "Direct",
"new_enrollmentdate": "2024-12-25T07:05:00Z"
}
```

Return Value (Used in Subsequent Steps):
```
{
"contactid": "550e8400-e29b-41d4-a716-446655440000" // UUID
}
```

Why Http Instead of Direct Connector:

- CRM API more flexible than native connector
- Can handle custom fields
- Easier error handling
- Better for complex business logic

Retry Policy:
```
{
"type": "exponential",
"count": 3,
"interval": "PT5S"
}
```

Why Retry:

- CRM availability varies during deployment
- Network timeouts common with cloud services
- Exponential backoff prevents service flooding
- Transient failures resolve quickly

When to Use:

- Creating records in external systems
- When native connector insufficient
- Complex transformation needed
- Error handling important

3.3 Generate Policy Number (Compose)

Purpose: Create unique policy identifier from enrollment data

Why Used:

- Policy number needed for multiple systems (billing, claims, ID card)
- Format: POL + Date + Member ID (unique and sortable)
- Generated here, used in multiple downstream actions
- Enables tracking of related records

Generation Formula:
POL + YYYYMMDD + First 8 chars of memberId
Example: POL20241225-550e8400

Why Compose Instead of SetVariable:

- Compose for one-time transformation
- Not reused later (unlike variables)
- More readable inline
- Compose is lightweight

When to Use:

- One-time data transformation
- Generating IDs/numbers
- Formatting output
- Lightweight computation

3.4 Http Action (Create Policy Record)

Purpose: Create policy record in CRM with generated policy number

Why Used:

- Policy record links member to coverage details
- Stores effective dates, plan type, premium info
- Required for claims processing to determine benefits
- Audit trail of policy creation

Policy Record:
{
"new_memberId": "550e8400-e29b-41d4-a716-446655440000",
"new_policynumber": "POL20241225-550e8400",
"new_policytype": "PPO Gold",
"new_effectivedate": "2025-01-01",

"new_status": "Active",
"new_enrollmentdate": "2024-12-25T07:05:00Z",
"new_dependentcount": 2
}

Why CRM and Not SQL:

- CRM policies are business objects with relationships
- May need audit trail (who created, when, changes)
- Portal may query CRM directly (not SQL)
- Future CRM customizations automatically included

When to Use:

- Creating business entities
- When audit trail/relationships matter
- Multi-system consistency needed
- External system is authoritative

3.5 Foreach (Create Dependent Records)

Purpose: Create contact records for each family member

Why Used:

- Family members need their own contacts for support
- Dependents must be in system for eligibility verification
- Relationship tracking for family deductible
- Enable dependent-specific communications

Dependent Data:
{
"firstName": "Jane",
"lastName": "Doe",
"dateOfBirth": "1992-03-20",
"relationship": "Spouse"
}

CRM Creation:
{
"firstname": "Jane",
"lastname": "Doe",
"birthdate": "1992-03-20",
"new_relationship": "Spouse",
"parentcontactid": "550e8400-e29b-41d4-a716-446655440000" // Link to primary member
}

Why Link to Parent:

- CRM tracks family relationships
- Family deductible calculated from parent + all dependents
- Portal shows dependent coverage with parent
- Simplifies family management

When to Use:

- Processing arrays of related items
- Each array item needs separate processing
- Order preservation important
- Related records need creation

3.6 Http Action (Generate ID Card)

Purpose: Create member ID card with barcode for claims submission

Why Used:

- Providers require ID card to verify coverage and eligibility
- Barcode enables quick lookup in provider system
- Card sent to member as physical proof
- Digital card also provided for mobile access

ID Card Service Input:
```
{
"memberId": "MEM-123456",
"firstName": "John",
"lastName": "Doe",
"policyNumber": "POL20241225-550e8400",
"effectiveDate": "2025-01-01",
"printFormat": "Physical"
}
```

Return Values:
```
{
"cardNumber": "IDC-123456789",
"cardImageUrl": "https://idcard-service.company.com/cards/IDC-123456789.png",
"barcodeData": "550e8400550e8400550e8400"
}
```

Why Separate Service:

- ID card generation is specialized (design, barcode, printing)
- External vendor manages print production
- Not part of enrollment workflow core logic
- Can scale independently

When to Use:

- Delegating specialized tasks
- Integrating with external vendors
- When third-party tool is best solution
- Separation of concerns

3.7 ApiConnection (SharePoint - Upload for Printing)

Purpose: Queue ID card image for printing and mailing

Why Used:

- Print vendor requires image file in specified format
- SharePoint list tracks print queue status
- Enables monitoring of card production
- Audit trail of cards printed

SharePoint List Columns:
{
"Title": "MEM-123456_IDCard",
"MemberId": "MEM-123456",
"CardImageURL": "https://idcard-service.company.com/cards/IDC-123456789.png",
"ShippingAddress": "123 Main St, New York, NY 10001",
"Status": "Ready for Print",
"CreatedDate": "2024-12-25T07:05:00Z"
}

Workflow:

1. Logic App creates list item (Ready for Print)
2. Print vendor polls SharePoint for new items
3. Vendor updates status (Printing → Shipped)
4. SharePoint triggers notification to member

Why SharePoint Instead of Email:

- Print vendor integrates with SharePoint
- List provides persistent queue
- Status tracking visible to operations
- Automation friendly

When to Use:

- Need persistent queue for external system
- Multi-step process tracking needed
- Status visibility important
- External system can poll for changes

3.8 Http Action (Create Portal Account)

Purpose: Create member portal login credentials

Why Used:

- Members access benefits, claims, eligibility online
- Portal account created automatically (no manual step)
- Temporary password sent separately for security
- Portal login starts with email verification

Portal API Request:
```
{
"memberId": "MEM-123456",
"email": "john.doe@email.com",
"firstName": "John",
"lastName": "Doe",
"generateTemporaryPassword": true
}
```

Portal API Response:
```
{
"accountId": "ACC-123456789",
"username": "john.doe@email.com",
"temporaryPassword": "TempPass123!@#",
"passwordExpiryDate": "2024-12-27T07:05:00Z",
"portalUrl": "https://member-portal.company.com"
}
```

Why Temporary Password:

- Member must set own password (security best practice)
- Email with temp password sent separately
- Member must change password on first login
- Prevents password interception in enrollment email

When to Use:

- Creating accounts in external systems
- Coordinating with authentication services
- When automated provisioning needed
- User activation workflows

3.9 Email Actions (Send Welcome Email)

Purpose: Communicate policy activation and portal access info

Why Used:

- Member needs confirmation of enrollment
- Portal credentials provided
- Expected timeline for ID card
- Sets expectations for coverage start date

Email Content:

Welcome, John!

Your policy has been successfully activated effective 2025-01-01

Policy Number: POL20241225-550e8400

Member ID: MEM-123456

Your ID card will arrive within 5-7 business days. You can access your account at our member portal

Temporary login credentials have been sent separately to this email.

Benefits effective: 2025-01-01

Why Email:

- Asynchronous communication (member reads when ready)
- Persistent record (can forward/share)
- Professional, templated appearance
- Compliance with communication requirements

When to Use:

- Notifying users of actions
- Providing credentials/documents
- Following up on requests
- Communication audit trail needed

3.10 Database Logging (SQL Insert)

Purpose: Record completed enrollment for analytics and audit

Why Used:

- Tracks enrollment success metrics
- Source data for reporting (enrollments per week, source analysis)
- Audit trail for compliance
- Enables troubleshooting of incomplete enrollments

Data Logged:

```
{
"memberId": "MEM-123456",
"firstName": "John",
"lastName": "Doe",
"policyNumber": "POL20241225-550e8400",
"enrollmentStatus": "Active",
"enrollmentDate": "2024-12-25T07:05:00Z",
"enrollmentSource": "Direct", // vs Broker, Employer
"idCardNumber": "IDC-123456789"
}
```

Queries Enabled:

```
-- Weekly enrollment report
SELECT COUNT(*), enrollmentSource
FROM Enrollments
WHERE enrollmentDate >= DATEADD(week, -1, GETDATE())
GROUP BY enrollmentSource

-- Members without ID cards
SELECT memberId, enrollmentDate
```

FROM Enrollments
WHERE idCardNumber IS NULL
AND enrollmentDate < DATEADD(day, -10, GETDATE())

When to Use:

- After successful completion
- For metrics/reporting
- Compliance/audit requirements
- Problem diagnosis (e.g., missing ID cards)

Project 4: Provider Network Management - Action Breakdown

Workflow Overview

SQL Trigger (Provider Update) → Extract Changes → Validate Contract
→ Query Networks → Retrieve Fee Schedule → Sync Systems
→ Update Directory → Log Success or Error

4.1 SQL Trigger (Modified Records)

Purpose: Automatically detect and respond to provider contract changes

Why Used:

- Manual notification of provider changes is unreliable
- System of record (SQL) changes drive workflows
- Real-time sync prevents stale provider data
- Automated response faster than manual process

Trigger Configuration:
-- Triggers on INSERT or UPDATE to Providers table
-- Detects: status changes, contract dates, specialty assignments

Triggered When:

- Provider status changes (Active → Inactive)
- Contract dates modified
- Fee schedules updated
- Credentialing status changes

When to Use:

- When source of truth is relational database
- Real-time response needed
- Frequent changes (contracts, rates)
- Multiple dependent systems

4.2 Validate Contract Status (If Action)

Purpose: Determine if provider contract currently active

Why Used:

- Only active providers included in network
- Ineligible providers: contract expired, credentialing incomplete
- Prevents customers from accessing terminated providers
- Enables automatic deactivation of inactive providers

Validation Criteria:
```
{
"and": [
{ "lessOrEquals": ["@contractStartDate", "@utcNow()"] },
{ "greaterOrEquals": ["@contractEndDate", "@utcNow()"] },
{ "equals": ["@credentialingStatus", "Approved"] }
]
}
```

Logic:

1. Contract start date <= Today (contract has started)
2. Contract end date >= Today (contract hasn't expired)
3. Credentialing complete (approved to see patients)

All conditions must be true for provider to be active.

When to Use:

- Date range validation
- Multi-condition approval
- Business rule evaluation
- Branching based on status

4.3 ApiConnection (Query Network Assignments)

Purpose: Find all insurance networks provider belongs to

Why Used:

- Providers often in multiple networks (HMO, PPO, Medicare, Medicaid)
- Each network has different fee schedules
- Claims must determine correct network for benefit calculation
- Network assignments enable proper routing

Query:
```
SELECT NetworkId, NetworkName, CoverageArea
FROM ProviderNetworkAssignments
WHERE ProviderId = '@{variables('providerId')}'
AND AssignmentStatus = 'Active'
```

Result Example:
```
[
{
"networkId": "NET-001",
"networkName": "Metro HMO Network",
"coverageArea": "New York Metro Area"
},
{
"networkId": "NET-002",
"networkName": "Regional PPO Network",
"coverageArea": "Northeast Region"
}
]
```

Why This Matters for Claims:

- Member checks in with provider
- Claims system queries provider network assignments
- Determines if in-network or out-of-network
- Applies correct copay/coinsurance

When to Use:

- Querying relationships (provider-networks)
- One-to-many lookups (provider in multiple networks)
- SQL joins needed
- Filtering results (WHERE clause)

4.4 Http Action (Sync to Eligibility System)

Purpose: Update eligibility system with new provider network assignments

Why Used:

- Eligibility service uses provider data for in-network determination
- Must be kept in sync with authoritative provider database
- Stale provider data = incorrect eligibility responses
- Real-time sync ensures accuracy

Sync Payload:

```
{
"providerId": "PROV-123456",
"providerName": "Johns Hospital",
"npi": "1234567890",
"primarySpecialty": "Hospital",
"secondarySpecialties": ["Emergency", "Cardiology"],
"networkAssignments": [
{ "networkId": "NET-001", "networkName": "Metro HMO" }
],
"feeSchedule": [
{ "procedureCode": "99213", "allowedAmount": 150 },
{ "procedureCode": "99214", "allowedAmount": 225 }
],
"status": "Active",
"lastUpdated": "2024-12-25T07:05:00Z"
}
```

Why Separate Sync:

- Eligibility service has its own database
- Different query patterns (lookup by provider for member eligibility)
- Performance optimized separately
- Can scale independently

Retry Policy Ensures Reliability:
```
{
"type": "exponential",
"count": 3,
"interval": "PT5S",
"maximumInterval": "PT30S"
}
```

When to Use:

- Syncing data between systems
- Keeping dependent systems updated
- Real-time integration important
- Multiple systems source from same data

4.5 CosmosDB Update Action

Purpose: Update provider directory document with latest status

Why Used:

- Provider directory searchable by members/providers
- Must reflect current network assignments
- Real-time updates improve accuracy
- CosmosDB provides global distribution for fast lookups

Update:
```
{
"status": "Active",
"lastSyncDate": "2024-12-25T07:05:00Z",
"networkAssignments": [
{ "networkId": "NET-001", "networkName": "Metro HMO" }
]
}
```

Why CosmosDB Instead of SQL:

- Global distribution for provider search queries
- Document structure matches provider data
- Real-time read replicas in multiple regions
- JSON-native storage

When to Use:

- Global distribution needed
- Hierarchical data structures
- Real-time read consistency
- Frequently searched data

4.6 Foreach (Contract Inactive Branch)

Purpose: Deactivate all network assignments when provider contract ends

Why Used:

- Provider termination must cascade through all networks
- Prevents patients from seeing terminated providers
- Maintains data integrity

Deactivation Actions:
UPDATE ProviderNetworkAssignments
SET AssignmentStatus = 'Inactive',
DeactivationDate = '@{utcNow()}'
WHERE ProviderId = '@{variables('providerId')}'
AND AssignmentStatus = 'Active'

Notification:
To: network-management@company.com
Subject: Provider Network Termination: Johns Hospital

Provider PROV-123456 contract has ended.
All network assignments have been marked inactive effective 2024-12-25.

When to Use:

- Cascading updates
- Cleanup operations
- Related data deactivation
- Maintaining referential integrity

4.7 Scope (Error Handling)

Purpose: Catch sync failures and log for investigation

Why Used:

- Network sync failures must be investigated
- Stale provider data breaks eligibility checks
- Operations team needs alerts
- Audit trail required for compliance

Error Logging:
{
"providerId": "PROV-123456",
"syncDate": "2024-12-25T07:05:00Z",
"syncStatus": "Failed",
"errorMessage": "Eligibility service returned 503 Service Unavailable",

"failedSystems": "Eligibility"
}

Alerting:
Teams Message to #provider-ops:
⚠ Provider Network Sync Failed
Provider: PROV-123456
Error: Eligibility service unavailable
Action: Retry in 5 minutes or escalate

When to Use:

- Critical operations (syncing data)
- When failures need investigation
- Operations monitoring required
- Compliance audit trails needed

Project 5: Claims Denial Appeals - Action Breakdown

Workflow Overview

Email Trigger → Parse Appeal → Validate Data → Retrieve Original Claim
→ Determine Level → Route to Reviewer → Create Task → Send Confirmation

5.1 Email Trigger

Purpose: Receive appeal submissions from members

Why Used:

- Members prefer email for appeals (asynchronous)
- Email provides documentation
- Can attach supporting documents
- Triggers workflow automatically

Email Parsing:
From: member@email.com
Subject: Appeal for Claim CLM-2024-001
Body:
I disagree with the denial of my claim for physical therapy.
The therapy was medically necessary for my back injury.

Attachments:

- doctor-letter.pdf (medical justification)
- prior-authorization.pdf

When to Use:

- Asynchronous input needed
- Documents may be attached
- User-friendly submission method
- Email audit trail valuable

5.2 ParseJson (Extract Appeal Details)

Purpose: Structure unstructured email content into typed fields

Why Used:

- Appeals arrive in various formats
- Email body text lacks structure
- Need to extract key information for routing
- Type safety prevents downstream errors

Extracted Fields:
```
{
"claimId": "CLM-2024-001",
"memberId": "MEM-123456",
"denialReason": "Not medically necessary",
"memberStatement": "Physical therapy was prescribed by my doctor...",
"attachedDocuments": ["doctor-letter.pdf", "auth.pdf"]
}
```

Parsing Rules:

- Extract claim ID from subject or body
- Extract member ID from email or body reference
- Extract member statement verbatim
- List attached document filenames

When to Use:

- Unstructured input (email, web forms)
- Need to convert to typed objects
- Multiple fields to extract
- Downstream actions need specific fields

5.3 Validate Appeal Eligibility (If Action)

Purpose: Ensure appeal includes required information

Why Used:

- Invalid appeals waste reviewer time
- Missing info prevents investigation
- Better to reject upfront than process incomplete appeal
- Sets expectations for members

Requirements:
```
{
"and": [
{ "not": { "equals": ["@claimId", ""] } },
{ "not": { "equals": ["@memberId", ""] } },
{ "not": { "equals": ["@memberStatement", ""] } }
]
}
```

Validation Failures:

- Return "Appeal invalid" response
- List missing required information
- No appeal created
- Member can resubmit with complete info

When to Use:

- Gate keeping before resource-intensive operations
- Validating user input
- Preventing invalid state creation
- Early failure (fail fast)

5.4 Query Original Claim (SQL)

Purpose: Retrieve original claim details for review

Why Used:

- Appeal decision depends on original claim details
- Claims database has complete claim history
- Including previous appeals enables context
- Enables consistency checking

Query:
```
SELECT ClaimId, MemberId, ClaimAmount, ApprovalAmount,
DenialReason, DenialDate, PreviousAppeals
FROM ClaimsProcessing
WHERE ClaimId = '@{claimId}'
```

Data Retrieved:
```
{
"claimId": "CLM-2024-001",
"claimAmount": 3000,
"denialReason": "Not medically necessary per medical policy",
"denialDate": "2024-11-15",
"previousAppeals": 0 // First appeal
}
```

Why Important:

- Determines appeal level (1st, 2nd, 3rd)
- Previous appeal context
- Original denial reasoning
- Helps reviewer prepare

When to Use:

- Context lookup for decisions
- Dependency on historical data
- Tracing decision history
- Pattern detection

5.5 Determine Appeal Level (Switch)

Purpose: Route appeal to appropriate level reviewer

Why Used:

- First appeals reviewed by benefits analysts
- Second appeals reviewed by physicians
- Third appeals reviewed by executives
- Level determines timeline and expertise

Logic:
If Previous Appeals = 0 → Level 1 (Benefits Analyst)
If Previous Appeals = 1 → Level 2 (Physician Reviewer)
If Previous Appeals = 2 → Level 3 (Executive)

Timeline Differences:

- **Level 1**: 30 days (expedited review)
- **Level 2**: 60 days (medical record review)
- **Level 3**: 30 days (executive discretion)

Escalation Pattern:

- Each level has higher authority
- Prevents duplicate review at same level
- Provides multiple opportunities to reverse denial

When to Use:

- Multi-level approval workflows
- Complexity determines routing
- Different experts for different levels
- Escalation patterns

5.6 Route to Reviewer (Query + Assignment)

Purpose: Find least busy reviewer at appropriate level

Why Used:

- Load balancing across review team
- Fair work distribution
- Prevents reviewer overload
- First response important for member satisfaction

Query for Level 1:
```
SELECT TOP 1 ReviewerId, ReviewerEmail
FROM ApprovalQueues
WHERE ReviewLevel = 1
AND QueueLength = (
SELECT MIN(QueueLength)
FROM ApprovalQueues
```

WHERE ReviewLevel = 1
)

Result: Reviewer with fewest pending appeals

Query for Level 2:
SELECT TOP 1 PhysicianId, PhysicianEmail
FROM MedicalReviewers
WHERE Specialty = '@{claimSpecialty}'
AND QueueLength = (
SELECT MIN(QueueLength)
FROM MedicalReviewers
WHERE Specialty IS NOT NULL
)

Result: Physician in relevant specialty with shortest queue

Why Specialty Matching for Level 2:

- Appeals for cardiology claims need cardiologist
- Relevant expertise improves decision quality
- Medical necessity determination requires specialty knowledge
- Consistency in review

When to Use:

- Load balancing needed
- Skill-based routing
- Queue management important
- Fair distribution

5.7 Create Appeal Record (SQL)

Purpose: Document appeal submission for audit trail

Why Used:

- Appeals must be logged per regulatory requirements
- Creates immutable record of appeal timeline
- Enables follow-up and status tracking
- Supports reporting on appeal rates/outcomes

Data Logged:
{
"appealId": "APPEAL20241225-CLM-2024-001",
"claimId": "CLM-2024-001",
"memberId": "MEM-123456",
"appealLevel": 1,
"appealStatus": "In Progress",
"submittedDate": "2024-12-25T07:05:00Z",
"reviewDeadline": "2025-01-24", // 30 days
"assignedReviewer": "analyst1@company.com",

"memberStatement": "Physical therapy was prescribed..."
}

Status Lifecycle:
Submitted → In Progress → Decision Pending → Approved/Denied → Closed

When to Use:

- Creating audit trails
- Regulatory compliance
- Status tracking
- Future analysis

5.8 Assign Task (Http to Approval System)

Purpose: Create review task in approval workflow system

Why Used:

- Reviewer has central task list
- Task includes deadlines and priority
- Enables workflow orchestration
- Prevents tasks from being forgotten

Task Request:
{
"appealId": "APPEAL20241225-CLM-2024-001",
"claimId": "CLM-2024-001",
"assignedTo": "analyst1@company.com",
"dueDate": "2025-01-24",
"appealLevel": 1,
"priority": "Normal", // Level 3 would be "Critical"
"description": "Review appeal for claim CLM-2024-001. Member states physical therapy was medically necessary.",
"attachments": ["doctor-letter.pdf", "prior-auth.pdf"]
}

Priority Rules:

- **Level 1**: Normal (30-day timeline)
- **Level 2**: High (60-day timeline, medical complexity)
- **Level 3**: Critical (executive review, likely time-sensitive)

When to Use:

- Creating tasks in external task systems
- Workflow management needed
- Team coordination required
- Deadline tracking important

5.9 Send Confirmation Email

Purpose: Notify member of appeal submission

Why Used:

- Members need confirmation their appeal received
- Reference number enables follow-up
- Timeline transparency improves satisfaction
- Professional communication expected

Confirmation Email:

Appeal Confirmation

Thank you for submitting your appeal.

Appeal ID: APPEAL20241225-CLM-2024-001

Claim ID: CLM-2024-001

Review Level: Level 1 (Benefits Analysis)

Review Deadline: January 24, 2025

We will contact you within 5 business days with updates on your appeal.

If you have questions, reply to this email.

Key Information:

- Appeal ID (for follow-up)
- Review timeline
- Contact information
- Expectation setting

When to Use:

- Confirming user submissions
- Providing reference numbers
- Setting expectations
- Professional communication

Common Action Patterns

Pattern 1: Query and Decision

Use When: Lookup followed by conditional logic

```
{
"Query": {
"type": "ApiConnection",
"inputs": { /* SQL query */ }
},
"Decision": {
"type": "If",
"expression": "@greater(length(body('Query')?['resultSets'][0]), 0)"
}
}
```

Example: Query member policy, then check if active

Pattern 2: Initialize, Transform, Update

Use When: Building complex objects across multiple steps

```
{
"Initialize": {
"type": "InitializeVariable"
},
"Transform": {
"type": "Compose"
},
"Update": {
"type": "SetVariable"
}
}
```

Example: Build claims response with multiple data sources

Pattern 3: Retry + Fallback

Use When: Call may fail temporarily

```
{
"type": "Http",
"inputs": { /* external API */ },
"retryPolicy": {
"type": "exponential",
"count": 3,
"interval": "PT5S"
}
}
```

Example: Call eligibility service with automatic retry

Pattern 4: Scope with Error Handler

Use When: Group related actions with specific error handling

```
{
"type": "Scope",
"actions": { /* main logic /}
},
{
"type": "Scope",
"runAfter": { "PreviousScope": ["Failed"] },
"actions": { / error handling */ }
}
```

Example: Calculate benefits, handle calculation errors separately

---

Pattern 5: Foreach with Conditional

Use When: Process array with condition per item

```
{
"type": "Foreach",
"foreach": "@triggerBody()?['items']",
"actions": {
"Check": {
"type": "If",
"expression": "@contains(variables('excluded'), item())"
}
}
}
```

Example: Check each procedure against exclusions

Decision Matrix: Choosing the Right Action

Data Retrieval

Scenario	Action	Why
Query SQL Server	ApiConnection (SQL)	Native connector, optimal for relational data
Query CosmosDB	ApiConnection (CosmosDB)	Native connector, JSON documents
Call external API	Http	Generic integration, supports any API
Call Azure service	ApiConnection	Native connector if available (faster)

Data Transformation

Scenario	Action	Why
One-time format change	Compose	Lightweight, reusable output
Update workflow state	SetVariable	Enables cross-action reference
Complex calculation	Scope (with actions)	Organized, error handling possible
Simple operation	Compose inline	No state needed

Decision Logic

Scenario	Action	Why
Single condition	If	Simple, readable
Multiple conditions	If (with 'and'/'or')	Groups related logic
Multiple branches	Switch	More than 2 paths
Multi-option logic	If with nested If	Complex decision tree

External Integration

Scenario	Action	Why
Custom API	Http	Generic, works with any API
Microsoft 365	ApiConnection	Native connector (Outlook, Teams)
Generic REST	Http	No connector available
Webhook style	Http trigger	Incoming webhook

Notifications

Scenario	Action	Why
Email notification	ApiConnection (Outlook)	Native, rich formatting
Teams message	ApiConnection (Teams)	Real-time visibility
Slack message	Http (Slack API)	No native connector
Event	ApiConnection (Event Grid)	Scalable pub-sub

Error Handling

Scenario	Action	Why
Specific action may fail	Add runAfter with ["Failed"]	Handles specific failures
Multiple actions may fail	Scope with error handler	Groups related error handling
Critical failure	Scope + Alert	Needs immediate escalation
Transient failure	Add retryPolicy	Automatic recovery

Document Version: 2.0
Last Updated: December 2025