12. - CRM Integration & Salesforce

Screen Pop: Architecture & Implementation

Overview

Screen pop is the automatic display of a customer record when they call. Agent's desktop automatically looks up the caller by phone number and displays their Salesforce record.

Benefits:

- No manual lookup needed (saves 30+ seconds per call)
- Agent sees customer context immediately
- Fewer wrong accounts selected
- Better first-call resolution

How It Works (The Flow)

```
Customer calls (ANI: +15551234567)
    ↓
Architect Flow receives call
    ↓
Data Action: Look up contact by phone
    in Salesforce API
    ↓
Salesforce returns: Contact ID + Account info
    ↓
Flow sets Interaction Attributes:
  - contact_id = "003xx000003SG"
  - account_id = "001xx000002Edc"
    ↓
Flow transfers to Support Queue
    ↓
```

```
Agent receives call
    ↓
Agent's desktop extension
    reads Interaction Attributes
    ↓
Desktop makes API call to Salesforce:
  GET /sobjects/Contact/003xx000003SG
    ↓
Browser pops Salesforce record in new window
    ↓
Agent sees: Name, account, cases, history
    ↓
Agent handles call with context
```

Architecture Components

1. Call Entry Point (Architect Flow)

The flow that receives inbound calls:

```
START
  ↓
Play: "Thank you for calling..."
  ↓
Data Action: lookup-contact-by-phone
  Input: ${interaction.caller.phoneNumber}
  Output: ${contact.id}, ${account.id}
  ↓
Decision: Contact found?
  YES → Set attributes → Transfer
  NO → Collect account number → Transfer
  ↓
Transfer to Queue
  (attributes go with call)
  ↓
END
```

2. Data Action (API Call)

```
{
  "name": "lookup-contact-by-phone",
  "method": "POST",
  "url": "https://your-instance.salesforce.com/services/apexrest/contact-lookup",
  "inputContract": {
    "phoneNumber": {
      "type": "string",
      "required": true
    }
  },
  "outputContract": {
    "success": { "type": "boolean" },
    "contactId": { "type": "string" },
    "firstName": { "type": "string" },
    "lastName": { "type": "string" },
    "accountId": { "type": "string" },
    "accountName": { "type": "string" },
    "tier": { "type": "string" }
  }
}
```

3. Interaction Attributes

Data attached to the call that agent's desktop can access:

```
{
  "contact_id": "003xx000003SG",
  "contact_name": "John Doe",
  "account_id": "001xx000002Edc",
  "account_name": "Acme Corp",
  "customer_tier": "Premium",
  "last_contact": "2026-03-10T14:30:00Z"
}
```

4. Agent Desktop Extension

JavaScript in the Genesys Workspace (desktop app or web):

```
// Listen for incoming interaction
genesysClient.on('interaction.incoming', async (interaction) => {
  // Get attributes set by flow
  const contactId = interaction.attributes?.contact_id;

  if (contactId) {
    // Pop Salesforce record
    const recordUrl = `https://your-instance.salesforce.com/${contactId}`;
    window.open(recordUrl, 'salesforce');
  }
});
```

Step-by-Step Implementation

Step 1: Create Salesforce Apex Endpoint

```
// ContactLookupController.cls

@RestResource(urlMapping='/contact-lookup')
global class ContactLookupController {
  @HttpPost
  global static LookupResponse lookup(String phoneNumber) {
    LookupResponse response = new LookupResponse();

    try {
      // Search for contact by phone
      List<Contact> contacts = [
        SELECT Id, FirstName, LastName, Email,
               AccountId, Account.Name,
               Custom_Tier__c
        FROM Contact
        WHERE Phone = :phoneNumber
           OR MobilePhone = :phoneNumber
        LIMIT 1
```

```
      ];

      if (contacts.isEmpty()) {
        response.success = false;
        return response;
      }

      Contact contact = contacts[0];
      response.success = true;
      response.contactId = contact.Id;
      response.firstName = contact.FirstName;
      response.lastName = contact.LastName;
      response.accountId = contact.AccountId;
      response.accountName = contact.Account?.Name;
      response.tier = contact.Custom_Tier__c;

    } catch (Exception e) {
      response.success = false;
      response.error = e.getMessage();
    }

    return response;
  }

  global class LookupResponse {
    public Boolean success;
    public String contactId;
    public String firstName;
    public String lastName;
    public String accountId;
    public String accountName;
    public String tier;
    public String error;
  }
}
```

Step 2: Create Architect Flow

In **Genesys Architect** → Create **Inbound Call Flow**:

Flow Steps:

```
1. START
   ↓
2. Play Audio: "Thank you for calling..."
   ↓
3. Data Action: lookup-contact-by-phone
   Input: ${interaction.caller.phoneNumber}
   ↓
4. Decision: ${dataAction.result.success}?

   IF YES:
   └─ Set Agent Variables:
      - contact_id = ${dataAction.result.contactId}
      - contact_name = ${dataAction.result.firstName} ${dataAction.result.lastName}
      - account_id = ${dataAction.result.accountId}
      - account_name = ${dataAction.result.accountName}
      - customer_tier = ${dataAction.result.tier}

   IF NO:
   └─ Play Audio: "Please hold while we locate your account..."

   ↓
5. Transfer to Queue: Support Queue
   ↓
6. DISCONNECT
```

Step 3: Create Desktop Extension

In **Genesys Cloud → Integrations → Custom Apps → Desktop App Extensions**:

```
// manifest.json
{
  "version": "1.0",
  "name": "Salesforce Screen Pop",
  "description": "Pops Salesforce contact on inbound call"
}

// index.html
```

```
<!DOCTYPE html>
<html>
<head>
  <script src="https://sdk.mypurecloud.com/v131/platform.min.js"></script>
</head>
<body>
  <script>
    const client = require('purecloud-platform-client-v2');

    // Initialize
    const platformClient = client.ApiClient.instance;
    platformClient.setEnvironment('mypurecloud.com');

    // Listen for incoming interaction
    const notificationService = platformClient.createNotificationService();

    notificationService.subscribe(
      'v2.conversations.{id}',
      async (event) => {
        const interaction = event.eventBody;

        // Check if attributes set by flow
        if (interaction.attributes?.contact_id) {
          const contactId = interaction.attributes.contact_id;
          const accountId = interaction.attributes.account_id;
          const contactName = interaction.attributes.contact_name;

          // Build Salesforce URL
          const baseUrl = 'https://your-instance.salesforce.com';
          const recordUrl = `${baseUrl}/${contactId}`;

          // Pop window
          window.open(recordUrl, 'salesforce_record',
            'width=1200,height=800,resizable=yes');

          console.log(`Screen pop: ${contactName} (${accountId})`);
        }
      }
    );
  </script>
```

```
</body>
</html>
```

Handling Edge Cases

Multiple Matches

Problem: Phone number found 3 contacts (different names, same company)

Solution: Let agent pick

```
Flow: Decision - ${dataAction.result.matches.length} > 1?

  IF YES:
  └─ Play: "Multiple accounts found. Press..."
  └─ Collect Input:
     "Press 1 for John Doe"
     "Press 2 for Jane Doe"
     "Press 3 for John Smith"
  └─ Set contact_id = ${dataAction.result.matches[input]}.id

  IF NO:
  └─ Continue normally
```

Contact Not Found

Problem: Phone number not in Salesforce

Solution: Offer fallback

```
Flow: Decision - ${dataAction.result.success}?

  IF NO:
  └─ Play: "Account not found. Please enter your account number..."
  └─ Collect Input: Account Number
  └─ Data Action: lookup-by-account-number
```

```
  └─ Set attributes if found
  └─ Transfer without pop if not found
```

Slow Lookup (Timeout)

Problem: Salesforce API slow, lookup takes 5+ seconds

Solution: Don't block the call

```
Flow: Data Action: lookup-contact-by-phone
  Timeout: 3 seconds

  Decision: Action succeeded?

    IF YES (within 3 sec):
    └─ Set attributes → Transfer

    IF NO (timed out):
    └─ Play: "Connecting you now..."
    └─ Transfer WITHOUT attributes
    └─ (Agent can manual search)
```

Salesforce Field Mapping

What agent sees after screen pop:

Data Point	Source	Salesforce Record
Contact Name	Flow attribute	Contact.Name
Email	API response	Contact.Email
Phone	API response	Contact.Phone
Account	Flow attribute	Account.Name
Tier/Priority	Flow attribute	Contact.Custom_Tier__c
Recent Cases	(automatic in SF)	Related Cases
Contact History	(automatic in SF)	Activity Timeline

Performance Optimization

Problem: Lookup taking 2+ seconds

Causes:

- Salesforce API slow
- Network latency
- SOQL query inefficient

Solutions:

1. **Add Index in Salesforce**

```
-- Make phone lookups fast
CREATE INDEX idx_contact_phone
ON Contact(Phone, MobilePhone);
```

2. **Cache recent lookups**

```
class ScreenPopCache {
  constructor(ttlSeconds = 3600) {
    this.cache = new Map();
    this.ttl = ttlSeconds;
  }

  async lookup(phone) {
    const cached = this.cache.get(phone);
    if (cached && Date.now() - cached.timestamp < this.ttl * 1000) {
      return cached.data;
    }

    const result = await callSalesforceAPI(phone);
    this.cache.set(phone, { data: result, timestamp: Date.now() });
    return result;
  }
}
```

3. **Use webhook instead of polling**
   - Salesforce creates contact → webhook updates Genesys
   - (More advanced, requires webhook setup)

Testing Screen Pop

Manual Test

1. Configure test flow in Architect
2. Set up desktop extension locally
3. Make test call to your Genesys DID
4. Verify:
   - ✓ Architect flow receives call
   - ✓ Data Action executes (check execution history)
   - ✓ Salesforce API returns contact
   - ✓ Flow sets interaction attributes
   - ✓ Desktop receives attributes
   - ✓ Window pops Salesforce record

Automated Test

```
// test-screen-pop.js

async function testScreenPop() {
  // 1. Test Salesforce lookup
  const contact = await lookupContactByPhone('+15551234567');
  console.assert(contact.id, 'Contact not found');

  // 2. Test Genesys Data Action
  const result = await callDataAction('lookup-contact-by-phone', {
    phoneNumber: '+15551234567'
  });
  console.assert(result.success, 'Data Action failed');

  // 3. Test flow
  const call = await makeTestCall('+15551234567');
  const attributes = call.attributes;
  console.assert(attributes.contact_id, 'No contact_id in attributes');

  console.log('✓ Screen pop test passed');
}
```

Troubleshooting

Problem: Desktop doesn't pop window

Check:

- ☐ Extension enabled in Genesys Workspace?
- ☐ Flow sets interaction attributes?
- ☐ Desktop JavaScript can access attributes?
- ☐ Browser allows popups?
- ☐ Salesforce URL correct?

Debug:

```
// Add logging to desktop extension
genesysClient.on('interaction.incoming', (interaction) => {
  console.log('Attributes:', interaction.attributes);
  console.log('Contact ID:', interaction.attributes?.contact_id);
});
```

Problem: Lookup returns "Contact not found" for valid phone

Check:

- ☐ Phone in Salesforce exactly matches flow phone?
- ☐ Phone format consistent (E.164, local, international)?
- ☐ Apex query correct?

Debug:

```
// Test in Salesforce Developer Console
List<Contact> contacts = [
  SELECT Id FROM Contact
```

```
  WHERE Phone = '+15551234567'
];
System.debug(contacts.size() + ' contacts found');
```

Problem: Performance is slow

Check:

- ☐ Data Action timeout too long?
- ☐ Salesforce API slow?
- ☐ Network latency high?

Optimize:

- Set timeout to 3 seconds (not 10)
- Add Salesforce index on Phone field
- Implement caching
- Use webhook instead of API call

Production Deployment Checklist

- ☐ Salesforce Apex endpoint created & tested
- ☐ Data Action configured in Genesys
- ☐ Architect flow configured & tested
- ☐ Desktop extension installed & enabled
- ☐ Error handling for timeouts
- ☐ Fallback for not-found contacts
- ☐ Field mapping verified
- ☐ Performance acceptable (<3 sec)
- ☐ Monitoring & alerting set up
- ☐ Staff trained on screen pop behavior

Related Topics

- Chapter 11: API Endpoints Reference (Salesforce API)
- Chapter 11: Error Handling & Retry Strategy
- Chapter 5: Data Actions (in Architect flows)
- Chapter 12: Contact Sync from Salesforce

Contact Sync Patterns

Overview

Contact sync keeps Genesys and your CRM (Salesforce, Dynamics, etc.) in sync. This guide covers two approaches:

1. **One-way sync** (CRM → Genesys, simpler, recommended first)
2. **Bi-directional sync** (both ways, complex, conflict resolution needed)

Strategy 1: One-Way Sync (CRM → Genesys)

Concept

Salesforce is the **master**. Genesys is a **read-only mirror**.

```
Salesforce (Master)
    ↓ (one direction)
Genesys (Mirror)

If contact changes in SF → update Genesys
If contact changes in Genesys → ignore (no write-back)
```

Pros

- ✅ Simpler (no conflict resolution)
- ✅ Faster (no locking)
- ✅ No race conditions
- ✅ Genesys data always matches SF

Cons

- □ Any changes in Genesys are lost on next sync
- □ Agents can't update CRM from Genesys

Sync Flow

```
Every 30 minutes:
  ↓
1. Query SF for changes:
   "SELECT * FROM Contact WHERE LastModifiedDate >= 30 min ago"
  ↓
2. For each contact:
   a. Check if exists in Genesys (by email, phone, external ID)
   b. If exists → PATCH (update only changed fields)
   c. If not exists → POST (create new)
  ↓
3. Log results (created, updated, skipped, errors)
  ↓
4. Alert if failures
```

Implementation

```
// sync-one-way.js

async function syncSalesforceToGenesys() {
  // 1. Fetch modified contacts from Salesforce
  const sfContacts = await querySalesforce(`
    SELECT Id, FirstName, LastName, Email, Phone
    FROM Contact
    WHERE LastModifiedDate >= LAST_N_MINUTES:30
  `);

  // 2. Get existing Genesys contacts
  const genesysContacts = await fetchGenesysContacts();
  const emailIndex = indexBy(genesysContacts, 'email');
```

```
  // 3. Process each SF contact
  let created = 0, updated = 0;

  for (const sfContact of sfContacts) {
    const genesysContact = emailIndex.get(sfContact.Email);

    if (genesysContact) {
      // Update existing
      await patchContact(genesysContact.id, {
        firstName: sfContact.FirstName,
        lastName: sfContact.LastName,
        phoneNumbers: [{ number: sfContact.Phone, type: 'WORK' }]
      });
      updated++;
    } else {
      // Create new
      await postContact({
        firstName: sfContact.FirstName,
        lastName: sfContact.LastName,
        email: sfContact.Email,
        phoneNumbers: [{ number: sfContact.Phone, type: 'WORK' }],
        externalId: sfContact.Id  // Link back
      });
      created++;
    }
  }

  console.log(`Created: ${created}, Updated: ${updated}`);
}
```

When to Use

- □ Initial implementation
- □ Salesforce is single source of truth
- □ Genesys only used for lookups/popups
- □ No need to update contacts from Genesys

Strategy 2: Bi-Directional Sync (Both Ways)

Concept

Both Salesforce AND Genesys can be updated. Sync runs both directions:

```
Salesforce ←→ Genesys


If SF updated → update Genesys
If Genesys updated → update Salesforce
If both updated at same time → conflict!
```

Pros

- ✅ Can update from both sides
- ✅ True two-way synchronization

Cons

- ❌ Much more complex
- ❌ Need conflict resolution
- ❌ Race conditions possible
- ❌ Slower (locking needed)
- ❌ Harder to debug

Conflict Resolution Strategies

Strategy A: Last-Write-Wins

Simple but risky:

```
Contact updated in both systems at same time:
  SF:     phone changed to 555-1111
  Genesys: phone changed to 555-2222
```

```
Winner: Whoever was modified LAST
Loser: Other change is overwritten
```

Problem: Unpredictable. User's change might be lost.

Strategy B: Field-Level Priority

Different fields have different sources:

```
contact_name:   SF always wins (SF is master for names)
phone_number:   Last-write-wins (allow edits in both)
tags:           Genesys always wins (agent tags)
```

Strategy C: Timestamp-Based with Locks

More robust:

```
1. Read contact with timestamp from both systems
2. Check: Who modified last?
   SF time > Genesys time → SF wins
   Genesys time > SF time → Genesys wins
3. Write winner's data to loser's system
4. Log conflict for human review
```

Implementation

```
// sync-bidirectional.js

async function syncBidirectional() {
  // 1. Fetch from both systems
  const sfContacts = await querySalesforce(`
    SELECT Id, FirstName, LastName, Email, LastModifiedDate
    FROM Contact
  `);

  const genesysContacts = await fetchGenesysContacts();

  // 2. Index for matching
```

```
const emailIndex = new Map();
for (const contact of [...sfContacts, ...genesysContacts]) {
  if (!emailIndex.has(contact.email)) {
    emailIndex.set(contact.email, []);
  }
  emailIndex.get(contact.email).push(contact);
}

// 3. Detect conflicts and sync
let conflicts = [];

for (const [email, records] of emailIndex) {
  if (records.length === 1) {
    // Only in one system, create in other
    await syncOneWay(records[0]);
  } else if (records.length === 2) {
    // In both systems, detect conflict
    const sf = records.find(r => r.system === 'salesforce');
    const gz = records.find(r => r.system === 'genesys');

    const winner = resolveConflict(sf, gz);

    if (winner === sf) {
      // SF wins, update Genesys
      await updateGenesys(gz.id, sf.data);
    } else {
      // Genesys wins, update SF
      await updateSalesforce(sf.id, gz.data);
    }

    // Log for audit
    if (sf.lastModifiedDate !== gz.dateModified) {
      conflicts.push({ email, sf, gz, winner });
    }
  }
}

// 4. Alert on conflicts
if (conflicts.length > 0) {
```

```
    await alertConflicts(conflicts);
  }
}

function resolveConflict(sfContact, gzContact) {
  // Last-write-wins
  const sfTime = new Date(sfContact.LastModifiedDate);
  const gzTime = new Date(gzContact.dateModified);

  if (sfTime > gzTime) {
    console.log(`Conflict: SF wins for ${sfContact.Email}`);
    return sfContact;
  } else {
    console.log(`Conflict: Genesys wins for ${sfContact.Email}`);
    return gzContact;
  }
}
```

Conflict Resolution Workflow

```
Conflict Detected
    ↓
Log both versions (SF and Genesys)
    ↓
Apply resolution strategy
    ├─ Last-write-wins?
    ├─ Field-level priority?
    └─ Manual approval?
    ↓
Update loser system
    ↓
Send alert to admin
    ↓
Update audit log
```

Data Deduplication

Problem: Same person, multiple records

```
Salesforce:
  Record 1: john.doe@example.com
  Record 2: JDoe@example.com (same person)

Genesys would create 2 records!
```

Solution: Match on Multiple Fields

```
function findMatchingContact(sfContact, genesysContacts) {
  // Try email (best match)
  let match = genesysContacts.find(
    gc => gc.email?.toLowerCase() === sfContact.Email.toLowerCase()
  );
  if (match) return match;

  // Try phone (second best)
  match = genesysContacts.find(
    gc => gc.phoneNumbers?.[0]?.number === sfContact.Phone
  );
  if (match) return match;

  // Try first + last name (risky, could be duplicate)
  match = genesysContacts.find(
    gc => gc.firstName === sfContact.FirstName &&
          gc.lastName === sfContact.LastName &&
          gc.externalOrganization?.id === sfContact.AccountId
  );
  if (match) return match;

  // Not found
  return null;
}
```

Best Practice: External IDs

Link records across systems:

```
Salesforce Contact:
  ID: 003xx000003SG
  Email: john@example.com

Genesys Contact:
  ID: contact-123
  externalId: "003xx000003SG"  ← Link back
  email: john@example.com
```

On next sync, use `externalId` to find exact match.

Field Mapping Reference

Salesforce	Genesys	Sync Direction	Conflicts
`Contact.Id`	`externalId`	SF → GZ	SF (master)
`Contact.FirstName`	`firstName`	Both	SF (master)
`Contact.LastName`	`lastName`	Both	SF (master)
`Contact.Email`	`email`	Both	Last-write
`Contact.Phone`	`phoneNumbers[0]`	Both	Last-write
`Contact.AccountId`	`org.externalId`	SF → GZ	SF
`Contact.LeadScore`	N/A	SF → GZ	N/A
`Contact.Tags` (custom)	`attributes.tags`	Both	GZ wins

Sync Frequency Trade-offs

Frequency	Pros	Cons	Use Case
Every 5 min	Real-time	High API load, cost	Critical data
Every 15 min	Near real-time	Medium load	Normal ops
Every 30 min	Balanced	Slight delay	Standard
Hourly	Low cost	1hr delay	Low priority

Frequency	Pros	Cons	Use Case
Daily	Very low cost	24hr delay	Batch jobs

Recommendation: Start with 30 minutes, adjust based on needs.

Monitoring Sync Health

```
class SyncMonitor {
  constructor() {
    this.history = [];
  }

  recordSync(result) {
    this.history.push({
      timestamp: new Date(),
      created: result.created,
      updated: result.updated,
      errors: result.errors,
      duration: result.duration,
      conflicts: result.conflicts
    });

    // Alert if too many errors
    if (result.errors.length > 10) {
      this.alertHighErrorRate(result);
    }

    // Alert if sync slow
    if (result.duration > 5 * 60 * 1000) { // 5 minutes
      this.alertSlowSync(result);
    }
  }

  getHealthScore() {
    const recent = this.history.slice(-10); // Last 10 syncs
    const avgErrors = recent.reduce((sum, h) => sum + h.errors.length, 0) / recent.length;
    const avgDuration = recent.reduce((sum, h) => sum + h.duration, 0) / recent.length;
```

```
    const score = 100 - (avgErrors * 5) - (avgDuration / 1000); // Deduct for errors and slow
    return Math.max(0, score);
  }
}
```

When to Use Which Strategy

One-Way Sync (Recommended First)

Use if:

- ✅ CRM is single source of truth
- ✅ Only need lookups in Genesys
- ✅ No updates from agent desktop
- ✅ Simple, maintainable

Bi-Directional Sync

Use if:

- ✅ Agents need to update contacts in Genesys
- ✅ Those updates must sync back to CRM
- ✅ Can handle complexity
- ✅ Have conflict resolution strategy

Migration Path

Phase 1: One-Way

- Sync SF → Genesys (read-only)
- Test with small set (100 contacts)
- Verify data accuracy

Phase 2: Expand

- Sync all contacts
- Run for 2 weeks, monitor
- Ensure no data loss

Phase 3: Bi-Directional

- Add reverse sync (Genesys → SF)
- Implement conflict resolution
- Test extensively
- Monitor for conflicts

Phase 4: Optimize

- Tune frequency
- Add caching
- Optimize performance
- Monitor cost

Related Topics

- Chapter 11: Real-World Contact Sync Example
- Chapter 11: API Endpoints Reference (Contacts API)
- Chapter 12: Activity Logging (logging interactions back)

Activity Logging & Webhooks

Overview

Activity logging means capturing what happened during a call and writing it back to your CRM. After the call ends, Genesys sends details to Salesforce so agents can track customer interactions.

Benefit: Single source of truth. All customer interactions visible in CRM.

The Flow

```
Call Happens
    ↓
Agent handles call (duration, queue, notes)
    ↓
Call ends
    ↓
Genesys webhook: conversation.ended
    ↓
Your backend receives webhook
    ↓
Fetch conversation details:
  - Caller phone
  - Duration
  - Agent name
  - Recording URL
  - Start/end time
    ↓
Find matching Salesforce contact (by phone)
    ↓
```

```
Create Salesforce Task:
  Subject: "Call with John Doe"
  Description: "Duration: 10 min..."
  Recording: [link]
  WhoId: Contact ID
  WhatId: Account ID
    ↓
Update Contact:
  LastActivityDate = today
  Last_Call_Date = today
    ↓
Success! Call logged in CRM
```

Setup: Genesys Webhook

In Genesys Admin

1. Go to **Integrations → Webhooks**
2. Click **Add Webhook**
3. Configure:
   - **Event:** `conversation.ended`
   - **URL:** `https://your-server.com/webhook/call-ended`
   - **Payload:** Send call details
   - **Retry:** 3 times if fails

Webhook Payload (What Genesys Sends)

```
{
  "conversationId": "conversation-111",
  "participantId": "user-agent-1",
  "conversationType": "phone",
  "callerId": "+15551234567",
  "calleeId": "+18001234567",
  "direction": "INBOUND",
  "startTime": "2026-03-14T10:00:00Z",
  "endTime": "2026-03-14T10:10:30Z",
```

```
  "durationSeconds": 630,
  "recordingId": "recording-222",
  "agentId": "user-agent-1",
  "agentName": "Agent Smith",
  "queueId": "queue-456",
  "queueName": "Support Queue",
  "interactionId": "interaction-123",
  "attributes": {
    "contact_id": "003xx000003SG",
    "account_id": "001xx000002Edc",
    "customer_tier": "Premium"
  }
}
```

Implementation: Webhook Handler

Node.js Server

```
const express = require('express');
const axios = require('axios');
require('dotenv').config();

const app = express();
app.use(express.json());

/**
 * Webhook endpoint - Genesys calls this when call ends
 */
app.post('/webhook/call-ended', async (req, res) => {
  const {
    conversationId,
    callerId,
    durationSeconds,
    recordingId,
    agentName,
    queueName,
```

```
    startTime,
    endTime,
    attributes
  } = req.body;

  console.log(`□□ Call ended: ${conversationId}, Duration: ${durationSeconds}s`);

  try {
    // 1. Find matching Salesforce contact
    const contact = await findSalesforceContactByPhone(callerId);

    if (!contact) {
      console.warn(`⚠ Contact not found for ${callerId}`);
      return res.status(200).json({ message: 'Contact not found' });
    }

    // 2. Get recording URL
    let recordingUrl = null;
    if (recordingId) {
      recordingUrl = await getRecordingUrl(recordingId);
    }

    // 3. Create Task in Salesforce
    const task = {
      Subject: `Call with ${contact.Name}`,
      Description: `
Inbound Call from ${callerId}
Duration: ${Math.floor(durationSeconds / 60)} minutes
Agent: ${agentName}
Queue: ${queueName}
Start: ${startTime}
End: ${endTime}
${recordingUrl ? `Recording: ${recordingUrl}` : ''}
      `.trim(),
      WhoId: contact.Id,          // Contact ID
      WhatId: contact.AccountId,  // Account ID
      ActivityDate: new Date().toISOString().split('T')[0],
      CallDurationInSeconds: durationSeconds,
      CallType: 'Inbound',
      Status: 'Completed',
```

```
      Type: 'Call'
    };

    const taskResult = await createSalesforceTask(task);
    console.log(`✅ Task created: ${taskResult.id}`);

    // 4. Update Contact's LastActivityDate
    await updateSalesforceContact(contact.Id, {
      LastActivityDate: new Date().toISOString().split('T')[0],
      Last_Call_Date__c: new Date().toISOString(),
      Last_Call_Duration__c: durationSeconds
    });

    res.status(200).json({ taskId: taskResult.id });

  } catch (error) {
    console.error('❌ Failed to log activity:', error.message);
    res.status(500).json({ error: error.message });
  }
});

/**
 * Find Salesforce contact by phone number
 */
async function findSalesforceContactByPhone(phoneNumber) {
  const query = `
    SELECT Id, Name, AccountId, Email
    FROM Contact
    WHERE Phone = '${phoneNumber}'
       OR MobilePhone = '${phoneNumber}'
    LIMIT 1
  `;

  const response = await axios.get(
    `${process.env.SALESFORCE_INSTANCE}/services/data/v57.0/query`,
    {
      params: { q: query },
      headers: {
        'Authorization': `Bearer ${process.env.SALESFORCE_TOKEN}`,
        'Content-Type': 'application/json'
```

```
      }
    }
  );

  return response.data.records[0] || null;
}

/**
 * Get Genesys recording URL
 */
async function getRecordingUrl(recordingId) {
  const response = await axios.get(
    `https://api.mypurecloud.com/api/v2/recordings/${recordingId}/media`,
    {
      headers: {
        'Authorization': `Bearer ${process.env.GENESYS_TOKEN}`
      },
      maxRedirects: 0,
      validateStatus: status => status === 307  // Expect redirect
    }
  );

  // Returns presigned S3 URL in Location header
  return response.headers.location || null;
}

/**
 * Create Salesforce Task
 */
async function createSalesforceTask(taskData) {
  const response = await axios.post(
    `${process.env.SALESFORCE_INSTANCE}/services/data/v57.0/sobjects/Task`,
    taskData,
    {
      headers: {
        'Authorization': `Bearer ${process.env.SALESFORCE_TOKEN}`,
        'Content-Type': 'application/json'
      }
    }
  );
```

```
  return response.data;
}

/**
 * Update Salesforce Contact
 */
async function updateSalesforceContact(contactId, updateData) {
  const response = await axios.patch(
    `${process.env.SALESFORCE_INSTANCE}/services/data/v57.0/sobjects/Contact/${contactId}`,
    updateData,
    {
      headers: {
        'Authorization': `Bearer ${process.env.SALESFORCE_TOKEN}`,
        'Content-Type': 'application/json'
      }
    }
  );

  return response.data;
}

const PORT = process.env.PORT || 3000;
app.listen(PORT, () => {
  console.log(`Webhook server running on port ${PORT}`);
});
```

Data Mapping

Call → Salesforce Task

Genesys Data	Salesforce Field	Mapping
`callerId` (phone)	Task subject	"Call with {contact.Name}"
`durationSeconds`	`CallDurationInSeconds`	Direct
`startTime`	Task description	Include timestamp

Genesys Data	Salesforce Field	Mapping
`endTime`	Task description	Include timestamp
`recordingId`	Task description	Link to recording
`agentName`	Task description	"Agent: {name}"
`queueName`	Task description	"Queue: {name}"
Contact ID (from lookup)	`WhoId`	Links to contact
Account ID (from lookup)	`WhatId`	Links to account
Current date	`ActivityDate`	Today's date

Advanced: Logging with Custom Fields

Salesforce custom fields capture more data:

```
// Salesforce custom fields setup
const task = {
  // Standard fields
  Subject: `Call with ${contact.Name}`,
  WhoId: contact.Id,
  WhatId: contact.AccountId,

  // Custom fields
  Call_Type__c: 'Inbound',
  Call_Queue__c: queueName,
  Call_Agent__c: agentName,
  Call_Duration_Seconds__c: durationSeconds,
  Call_Recording_URL__c: recordingUrl,
  Call_Channel__c: 'Phone',
  Customer_Tier__c: attributes?.customer_tier,
  Was_Transferred__c: false,
  Call_Outcome__c: 'Completed',
  Agent_Notes__c: attributes?.notes
};
```

```
// Custom field names end with __c (Salesforce convention)
```

Error Handling

Problem: Contact Not Found

```
// Option 1: Skip (don't log if no contact)
if (!contact) {
  console.warn(`No contact for ${callerId}`);
  return res.status(200).json({ message: 'Contact not found' });
}

// Option 2: Create contact
if (!contact) {
  contact = await createSalesforceContact({
    LastName: callerId,  // Use phone as fallback
    Phone: callerId
  });
}

// Option 3: Log to generic "unknown caller" account
if (!contact) {
  contact = { Id: UNKNOWN_CALLER_ACCOUNT };
}
```

Problem: Recording URL Timeout

```
async function getRecordingUrlSafe(recordingId, timeoutMs = 3000) {
  try {
    const response = await axios.get(
      `https://api.mypurecloud.com/api/v2/recordings/${recordingId}/media`,
      {
        headers: { 'Authorization': `Bearer ${token}` },
        timeout: timeoutMs
      }
```

```
    );
    return response.headers.location || null;
  } catch (error) {
    if (error.code === 'ECONNABORTED') {
      console.warn('Recording URL timeout');
      return null;  // Don't block task creation
    }
    throw error;
  }
}
```

Problem: Task Creation Fails

```
async function createTaskWithRetry(task, maxAttempts = 3) {
  for (let attempt = 1; attempt <= maxAttempts; attempt++) {
    try {
      const result = await createSalesforceTask(task);
      console.log(`  Task created: ${result.id}`);
      return result;
    } catch (error) {
      if (attempt === maxAttempts) {
        console.error(`  Failed after ${maxAttempts} attempts`);
        throw error;
      }

      const delayMs = 1000 * Math.pow(2, attempt - 1); // Exponential backoff
      console.warn(`Retry in ${delayMs}ms...`);
      await sleep(delayMs);
    }
  }
}
```

Webhook Security

Verify Genesys Signature

Genesys signs webhooks so you can verify they're legitimate:

```
const crypto = require('crypto');

/**
 * Verify Genesys webhook signature
 */
function verifyWebhookSignature(request, secret) {
  const signature = request.headers['x-genesys-webhook-signature'];
  const timestamp = request.headers['x-genesys-webhook-timestamp'];

  if (!signature || !timestamp) {
    throw new Error('Missing signature or timestamp header');
  }

  // Reconstruct the signed string
  const body = request.rawBody; // Must be raw, not parsed
  const signedString = `${timestamp}.${body}`;

  // Calculate HMAC
  const hash = crypto
    .createHmac('sha256', secret)
    .update(signedString)
    .digest('base64');

  // Verify
  if (hash !== signature) {
    throw new Error('Webhook signature invalid');
  }

  return true;
}

// Middleware to verify all webhooks
app.use((req, res, next) => {
  req.rawBody = req.body;  // Keep raw body
  next();
});

app.post('/webhook/call-ended', (req, res, next) => {
```

```
  try {
    verifyWebhookSignature(req, process.env.GENESYS_WEBHOOK_SECRET);
    next();
  } catch (error) {
    console.error('❌ Webhook signature verification failed:', error.message);
    return res.status(401).json({ error: 'Unauthorized' });
  }
});
```

Monitoring & Alerts

```
class ActivityLoggingMonitor {
  constructor() {
    this.stats = {
      totalCalls: 0,
      logsCreated: 0,
      logsFailed: 0,
      contactsNotFound: 0,
      avgProcessingTime: 0
    };
  }

  recordSuccess(processingTimeMs) {
    this.stats.totalCalls++;
    this.stats.logsCreated++;
    this.updateAvgTime(processingTimeMs);
  }

  recordFailure(reason) {
    this.stats.totalCalls++;
    this.stats.logsFailed++;

    if (reason === 'contact_not_found') {
      this.stats.contactsNotFound++;
    }

    // Alert if too many failures
```

```
    const failureRate = this.stats.logsFailed / this.stats.totalCalls;
    if (failureRate > 0.05) { // > 5%
      this.alertHighFailureRate(failureRate);
    }
  }

  updateAvgTime(newTime) {
    const count = this.stats.logsCreated;
    this.stats.avgProcessingTime =
      (this.stats.avgProcessingTime * (count - 1) + newTime) / count;
  }

  reportHealth() {
    return `
Activity Logging Health:
  Total calls: ${this.stats.totalCalls}
  Logged: ${this.stats.logsCreated}
  Failed: ${this.stats.logsFailed}
  Not found: ${this.stats.contactsNotFound}
  Avg time: ${this.stats.avgProcessingTime.toFixed(0)}ms
  Success rate: ${((this.stats.logsCreated / this.stats.totalCalls) * 100).toFixed(1)}%
    `;
  }
}
```

Production Checklist

- ☐ Genesys webhook configured for `conversation.ended`
- ☐ Webhook URL is publicly accessible & secured
- ☐ Signature verification implemented
- ☐ Error handling for missing contacts
- ☐ Recording URL generation working
- ☐ Salesforce Task fields mapped
- ☐ Contact lookup by phone working
- ☐ Task creation tested with real calls
- ☐ Monitoring & alerting set up

- [ ] Staff trained (calls logged to CRM)
- [ ] Documentation updated

Related Topics

- Chapter 12: Screen Pop (related feature)
- Chapter 12: Contact Sync Patterns (keeping contacts in sync)
- Chapter 11: API Endpoints Reference (Genesys APIs)

Bi-Directional Sync with Conflict Resolution

Overview

Bi-directional sync means changes flow both ways: Salesforce → Genesys AND Genesys → Salesforce. This is more complex than one-way sync because conflicts can occur when both systems are updated simultaneously.

Conflict Example:

```
Same contact updated at same time:
  Salesforce: phone changed to +15551111111 at 10:00:05
  Genesys: phone changed to +15552222222 at 10:00:06

Question: Which phone number wins?
```

Conflict Resolution Strategies

Strategy 1: Last-Write-Wins (Simplest)

The system modified MOST RECENTLY wins:

```
function resolveConflict(sfContact, gzContact) {
  const sfTime = new Date(sfContact.LastModifiedDate).getTime();
  const gzTime = new Date(gzContact.dateModified).getTime();

  if (sfTime > gzTime) {
    console.log(`Conflict: SF wins (${sfTime} > ${gzTime})`);
    return { winner: 'salesforce', action: 'update_genesys' };
  } else {
```

```
    console.log(`Conflict: Genesys wins (${gzTime} >= ${sfTime})`);
    return { winner: 'genesys', action: 'update_salesforce' };
  }
}
```

Pros:

- Simple
- No manual intervention
- Works for most cases

Cons:

- Unpredictable (users don't know who wins)
- Possible data loss
- No audit trail

Strategy 2: Field-Level Priority

Different fields have different masters:

```
const fieldPriority = {
  // Names are always SF (master data)
  'firstName': 'salesforce',
  'lastName': 'salesforce',

  // Phone can be updated in either (last-write-wins)
  'phoneNumbers': 'last_write',
  'email': 'last_write',

  // Tags are Genesys (agent annotations)
  'tags': 'genesys',
  'notes': 'genesys',

  // Organization always SF
  'organization': 'salesforce'
};

function applyFieldLevelResolution(sfContact, gzContact) {
  const merged = {};
```

```
  for (const field in fieldPriority) {
    const priority = fieldPriority[field];

    if (priority === 'salesforce') {
      // Always use SF value
      merged[field] = sfContact[field];
    } else if (priority === 'genesys') {
      // Always use Genesys value
      merged[field] = gzContact[field];
    } else if (priority === 'last_write') {
      // Use whoever was modified last
      const sfTime = sfContact.LastModifiedDate || 0;
      const gzTime = gzContact.dateModified || 0;
      merged[field] = new Date(sfTime) > new Date(gzTime)
        ? sfContact[field]
        : gzContact[field];
    }
  }

  return merged;
}
```

Pros:

- Logical (names stay in SF, notes stay in GZ)
- Flexible
- Reduces conflicts

Cons:

- More complex
- Requires configuration
- Still possible data loss

Strategy 3: Timestamp-Based with Locking

Most robust: lock during sync, use timestamps, log everything:

```
async function syncWithLocking(sfContact, gzContact) {
  // 1. Lock both records (prevent other changes during sync)
  await lockSalesforceRecord(sfContact.Id);
  await lockGenesysRecord(gzContact.id);

  try {
    // 2. Detect if either was modified since we last synced
    const lastSyncTime = new Date(sfContact.Last_Sync__c);
    const sfModified = new Date(sfContact.LastModifiedDate) > lastSyncTime;
    const gzModified = new Date(gzContact.dateModified) > lastSyncTime;

    if (!sfModified && !gzModified) {
      // No changes, nothing to do
      return { status: 'no_change' };
    }

    if (sfModified && !gzModified) {
      // Only SF changed, update Genesys
      await updateGenesysContact(gzContact.id, sfContact);
      return { status: 'sf_updated_gz' };
    }

    if (!sfModified && gzModified) {
      // Only Genesys changed, update SF
      await updateSalesforceContact(sfContact.Id, gzContact);
      return { status: 'gz_updated_sf' };
    }

    // CONFLICT: Both changed
    const resolution = await resolveConflictWithLogging(sfContact, gzContact);

    if (resolution.winner === 'salesforce') {
      await updateGenesysContact(gzContact.id, sfContact);
    } else {
      await updateSalesforceContact(sfContact.Id, gzContact);
    }

    // 3. Log conflict for audit
    await logConflict({
      timestamp: new Date(),
```

```
      contactId: sfContact.Id,
      sfLastModified: sfContact.LastModifiedDate,
      gzLastModified: gzContact.dateModified,
      winner: resolution.winner,
      differences: resolution.differences,
      action: resolution.action
    });

    return { status: 'conflict_resolved', winner: resolution.winner };

  } finally {
    // 4. Unlock records
    await unlockSalesforceRecord(sfContact.Id);
    await unlockGenesysRecord(gzContact.id);
  }
}
```

Pros:

- Most reliable
- Prevents race conditions
- Complete audit trail
- Can alert on conflicts

Cons:

- Complex
- Slower (locking overhead)
- Requires timestamp tracking

Implementation: Full Bi-Directional Sync

```
class BidirectionalSync {
  constructor(config) {
    this.config = config;
    this.stats = {
      synced: 0,
```

```
    conflicts: 0,
    errors: 0
  };
  this.conflicts = [];
}

/**
 * Main sync method
 */
async runSync() {
  console.log('\n=== BIDIRECTIONAL SYNC START ===');

  try {
    // 1. Fetch from both systems
    const sfContacts = await this.fetchAllSalesforceContacts();
    const gzContacts = await this.fetchAllGenesysContacts();

    console.log(`Salesforce: ${sfContacts.length} contacts`);
    console.log(`Genesys: ${gzContacts.length} contacts`);

    // 2. Index for matching
    const sfById = new Map(sfContacts.map(c => [c.Id, c]));
    const sfByEmail = new Map(sfContacts.map(c => [c.Email?.toLowerCase(), c]));
    const gzByExtId = new Map(gzContacts.map(c => [c.externalId, c]));

    // 3. Find and sync all contact pairs
    const synced = new Set();

    // Process Salesforce contacts
    for (const sf of sfContacts) {
      let gz = gzByExtId.get(sf.Id) // Match by externalId first
            || sfByEmail.get(sf.Email?.toLowerCase()); // Then by email

      if (gz) {
        // Both systems have it
        await this.syncPair(sf, gz);
        synced.add(sf.Id);
      } else {
        // Only in Salesforce, create in Genesys
        await this.createInGenesys(sf);
```

```
        synced.add(sf.Id);
      }
    }

    // Process Genesys-only contacts
    for (const gz of gzContacts) {
      if (!synced.has(gz.externalId)) {
        // Only in Genesys, create in Salesforce
        await this.createInSalesforce(gz);
      }
    }

    this.printResults();

  } catch (error) {
    console.error('❌ SYNC FAILED:', error);
    throw error;
  }
}

/**
 * Sync a contact that exists in both systems
 */
async syncPair(sfContact, gzContact) {
  try {
    // Detect changes
    const sfModified = this.isModifiedSinceLast(sfContact);
    const gzModified = this.isModifiedSinceLast(gzContact);

    if (!sfModified && !gzModified) {
      // No changes
      return;
    }

    if (sfModified && !gzModified) {
      // SF changed, update Genesys
      await this.updateGenesys(gzContact.id, sfContact);
      this.stats.synced++;
      return;
    }
```

```
    if (!sfModified && gzModified) {
      // Genesys changed, update SF
      await this.updateSalesforce(sfContact.Id, gzContact);
      this.stats.synced++;
      return;
    }

    // CONFLICT: Both changed
    await this.handleConflict(sfContact, gzContact);

  } catch (error) {
    console.error(`Error syncing ${sfContact.Id}:`, error);
    this.stats.errors++;
  }
}

/**
 * Handle conflict between SF and Genesys
 */
async handleConflict(sfContact, gzContact) {
  const resolution = this.resolveConflict(sfContact, gzContact);

  console.warn(`⚠ CONFLICT for ${sfContact.Email}:`);
  console.warn(`   SF: ${sfContact.LastModifiedDate}`);
  console.warn(`   GZ: ${gzContact.dateModified}`);
  console.warn(`   Winner: ${resolution.winner}`);

  // Apply resolution
  if (resolution.winner === 'salesforce') {
    await this.updateGenesys(gzContact.id, sfContact);
  } else {
    await this.updateSalesforce(sfContact.Id, gzContact);
  }

  // Log for review
  this.conflicts.push({
    email: sfContact.Email,
    sfTime: sfContact.LastModifiedDate,
    gzTime: gzContact.dateModified,
```

```
    winner: resolution.winner,
    changes: resolution.changes
  });

  this.stats.conflicts++;
  this.stats.synced++;
}

/**
 * Resolve conflict (strategy: last-write-wins with field priority)
 */
resolveConflict(sfContact, gzContact) {
  const fieldPriority = {
    firstName: 'sf',
    lastName: 'sf',
    email: 'last_write',
    phoneNumbers: 'last_write',
    externalOrganization: 'sf',
    tags: 'gz',
    attributes: 'last_write'
  };

  const sfTime = new Date(sfContact.LastModifiedDate).getTime();
  const gzTime = new Date(gzContact.dateModified).getTime();
  const overallWinner = sfTime > gzTime ? 'salesforce' : 'genesys';

  const merged = {};
  const changes = {};

  for (const field in fieldPriority) {
    const priority = fieldPriority[field];
    let value;

    if (priority === 'sf') {
      value = sfContact[field];
    } else if (priority === 'gz') {
      value = gzContact[field];
    } else {
      // last_write
      value = sfTime > gzTime ? sfContact[field] : gzContact[field];
```

```
    }

    if (sfContact[field] !== gzContact[field]) {
      changes[field] = {
        sf: sfContact[field],
        gz: gzContact[field],
        winner: priority === 'sf' ? 'sf' : priority === 'gz' ? 'gz' : overallWinner
      };
    }

    merged[field] = value;
  }

  return {
    winner: overallWinner,
    merged,
    changes
  };
}

/**
 * Check if contact was modified since last sync
 */
isModifiedSinceLast(contact) {
  if (contact.Last_Sync__c) {
    const lastSync = new Date(contact.Last_Sync__c).getTime();
    const lastMod = new Date(contact.LastModifiedDate || contact.dateModified).getTime();
    return lastMod > lastSync;
  }
  return true; // Assume modified if no last sync
}

/**
 * Update Genesys with SF data
 */
async updateGenesys(gzId, sfContact) {
  const updateData = {
    firstName: sfContact.FirstName,
    lastName: sfContact.LastName,
    email: sfContact.Email,
```

```
    phoneNumbers: sfContact.Phone ?
      [{ number: sfContact.Phone, type: 'WORK' }] : [],
    externalId: sfContact.Id  // Keep the link
  };

  await axios.patch(
    `https://api.mypurecloud.com/api/v2/contacts/${gzId}`,
    updateData,
    { headers: { 'Authorization': `Bearer ${this.gzToken}` } }
  );

  console.log(`  Updated Genesys: ${gzId}`);
}

/**
 * Update Salesforce with Genesys data
 */
async updateSalesforce(sfId, gzContact) {
  const updateData = {
    FirstName: gzContact.firstName,
    LastName: gzContact.lastName,
    Email: gzContact.email,
    Phone: gzContact.phoneNumbers?.[0]?.number,
    Last_Sync__c: new Date().toISOString()
  };

  await axios.patch(
    `${process.env.SALESFORCE_INSTANCE}/services/data/v57.0/sobjects/Contact/${sfId}`,
    updateData,
    { headers: { 'Authorization': `Bearer ${this.sfToken}` } }
  );

  console.log(`  Updated Salesforce: ${sfId}`);
}

/**
 * Create in Genesys (from Salesforce)
 */
async createInGenesys(sfContact) {
  const newContact = {
```

```
    firstName: sfContact.FirstName,
    lastName: sfContact.LastName,
    email: sfContact.Email,
    phoneNumbers: sfContact.Phone ?
      [{ number: sfContact.Phone, type: 'WORK' }] : [],
    externalId: sfContact.Id
  };

  const response = await axios.post(
    `https://api.mypurecloud.com/api/v2/contacts`,
    newContact,
    { headers: { 'Authorization': `Bearer ${this.gzToken}` } }
  );

  console.log(`  Created in Genesys: ${response.data.id}`);
  this.stats.synced++;
}

/**
 * Create in Salesforce (from Genesys)
 */
async createInSalesforce(gzContact) {
  const newContact = {
    FirstName: gzContact.firstName,
    LastName: gzContact.lastName,
    Email: gzContact.email,
    Phone: gzContact.phoneNumbers?.[0]?.number,
    Last_Sync__c: new Date().toISOString()
  };

  const response = await axios.post(
    `${process.env.SALESFORCE_INSTANCE}/services/data/v57.0/sobjects/Contact`,
    newContact,
    { headers: { 'Authorization': `Bearer ${this.sfToken}` } }
  );

  // Update Genesys with SF ID (externalId)
  await this.updateGenesys(gzContact.id, { ...gzContact, Id: response.data.id });

  console.log(`  Created in Salesforce: ${response.data.id}`);
```

```
    this.stats.synced++;
  }

  /**
   * Print results
   */
  printResults() {
    console.log('\n=== SYNC RESULTS ===');
    console.log(`✓ Synced: ${this.stats.synced}`);
    console.log(`⚠ Conflicts: ${this.stats.conflicts}`);
    console.log(`✗ Errors: ${this.stats.errors}`);

    if (this.conflicts.length > 0) {
      console.log('\nConflicts (for review):');
      this.conflicts.forEach(c => {
        console.log(`  ${c.email}: ${c.winner} won`);
      });
    }
  }
}

// Usage
const sync = new BidirectionalSync(config);
await sync.runSync();
```

Deduplication Across Systems

By External ID (Best)

```
// Each contact has ID from the other system
Salesforce Contact:
  Id: 003xx000003SG
  External_Genesys_ID__c: "contact-123"

Genesys Contact:
  Id: contact-123
```

```
  externalId: "003xx000003SG"

// Match on these, never duplicate
```

By Email (Good)

```
const sfContact = sfContacts.find(c => c.Email === gzContact.email);

// Assumes email is unique per contact
// Risk: Same person with multiple emails
```

By Phone (Risky)

```
const sfContact = sfContacts.find(c => c.Phone === gzContact.phoneNumbers[0]?.number);

// Risk: Multiple people sharing phone (family, shared line)
```

Monitoring & Alerts

```
class SyncMonitor {
  constructor() {
    this.history = [];
  }

  recordSync(result) {
    this.history.push({
      timestamp: new Date(),
      synced: result.synced,
      conflicts: result.conflicts,
      errors: result.errors
    });

    // Alert if many conflicts
    if (result.conflicts > 10) {
      console.warn(`⚠ HIGH CONFLICT RATE: ${result.conflicts} conflicts`);
```

```
      this.alertOps(result);
    }

    // Alert if errors
    if (result.errors > 5) {
      console.error(`🚨 HIGH ERROR RATE: ${result.errors} errors`);
      this.alertOps(result);
    }
  }

  getConflictRate() {
    const recent = this.history.slice(-10);
    const totalConflicts = recent.reduce((sum, h) => sum + h.conflicts, 0);
    const totalSynced = recent.reduce((sum, h) => sum + h.synced, 0);
    return totalConflicts / totalSynced;
  }
}
```

Best Practices

1. **Add Last_Sync timestamp** to both systems
   - Used to detect what changed since last sync
   - Needed for conflict detection
2. **Use External IDs**
   - SF: `External_Genesys_ID__c`
   - Genesys: `externalId`
   - Primary way to match contacts
3. **Log all conflicts**
   - Build audit trail
   - Manual review capability
   - Alerts ops team
4. **Test thoroughly**
   - What if SF and GZ both update same field?
   - What if network fails mid-sync?
   - What if token expires?
5. **Start one-way, migrate to bi-directional**
   - Less risk
   - Easier to debug
   - Can add complexity later

Related Topics

- Chapter 12: Contact Sync Patterns (one-way alternative)
- Chapter 12: Activity Logging (logging sync conflicts)
- Chapter 11: Error Handling & Retry Strategy

GDPR & Data Governance in CRM Integration

Overview

When syncing contact data between Genesys and Salesforce, you handle personal data (PII - Personally Identifiable Information). GDPR, CCPA, and similar regulations require proper handling, deletion, and consent management.

Key Regulations:

- **GDPR (EU):** Right to be forgotten, consent required
- **CCPA (California):** Right to deletion, right to know
- **PIPEDA (Canada):** Similar to GDPR
- **Your Local Laws:** May have additional requirements

PII Data Types

What's PII?

Information that can identify a person:

```
✓ PII - Don't store longer than needed:
  - Full name
  - Email address
  - Phone number
  - Address
  - Social Security Number
  - Payment card info
  - Biometric data
  - IP address + timestamp
```

```
✗ NOT PII - Can store longer:
  - Company name
  - Job title
  - Aggregated analytics (no individuals)
  - Anonymized data (truly de-identified)
```

GDPR Right to Deletion ("Right to be Forgotten")

Requirement

Customer asks: "Delete all my data"

You MUST:

1. Delete from Salesforce
2. Delete from Genesys
3. Delete from call recordings
4. Delete from transcripts
5. Delete from logs/backups (after retention period)

Implementation

```
/**
 * GDPR: Delete all data for a contact
 */
async function deleteContactCompletelyGDPR(email) {
  console.log(`□□ GDPR Deletion: ${email}`);

  try {
    // 1. Find contact in both systems
    const sfContact = await findSalesforceContactByEmail(email);
    const gzContact = await findGenesysContactByEmail(email);

    if (!sfContact && !gzContact) {
```

```
  console.log(`  No contact found for ${email}`);
  return { status: 'not_found' };
}

// 2. Delete from Salesforce
if (sfContact) {
  console.log(`  Deleting Salesforce contact: ${sfContact.Id}`);
  await axios.delete(
    `${process.env.SALESFORCE_INSTANCE}/services/data/v57.0/sobjects/Contact/${sfContact.Id}`,
    { headers: { 'Authorization': `Bearer ${this.sfToken}` } }
  );

  // Also delete associated records
  await deleteAssociatedSalesforceRecords(sfContact.Id);
}

// 3. Delete from Genesys
if (gzContact) {
  console.log(`  Deleting Genesys contact: ${gzContact.id}`);
  await axios.delete(
    `https://api.mypurecloud.com/api/v2/contacts/${gzContact.id}`,
    { headers: { 'Authorization': `Bearer ${this.gzToken}` } }
  );
}

// 4. Find and delete recordings (if applicable)
const recordings = await findRecordingsByPhone(sfContact?.Phone);
for (const recording of recordings) {
  console.log(`  Deleting recording: ${recording.id}`);
  await deleteRecording(recording.id);
}

// 5. Find and delete call logs
const callLogs = await findCallLogsByEmail(email);
for (const log of callLogs) {
  console.log(`  Deleting call log: ${log.id}`);
  await deleteCallLog(log.id);
}

// 6. Log the deletion (for compliance audit)
```

```
    await logGDPRDeletion({
      email,
      deletedAt: new Date(),
      deletedFrom: ['salesforce', 'genesys', 'recordings', 'call_logs'],
      reason: 'GDPR Right to Deletion'
    });

    console.log(`✅ Completely deleted: ${email}`);
    return { status: 'deleted', deletedFrom: ['salesforce', 'genesys'] };

  } catch (error) {
    console.error(`❌ Deletion failed: ${error.message}`);
    throw new Error(`Failed to delete ${email}: ${error.message}`);
  }
}

/**
 * Delete associated Salesforce records (tasks, events, etc.)
 */
async function deleteAssociatedSalesforceRecords(contactId) {
  // Delete tasks
  const tasks = await querySalesforce(`
    SELECT Id FROM Task
    WHERE WhoId = '${contactId}'
  `);

  for (const task of tasks.records) {
    await axios.delete(
      `${process.env.SALESFORCE_INSTANCE}/services/data/v57.0/sobjects/Task/${task.Id}`,
      { headers: { 'Authorization': `Bearer ${this.sfToken}` } }
    );
  }

  // Delete events
  const events = await querySalesforce(`
    SELECT Id FROM Event
    WHERE WhoId = '${contactId}'
  `);

  for (const event of events.records) {
```

```
    await axios.delete(
      `${process.env.SALESFORCE_INSTANCE}/services/data/v57.0/sobjects/Event/${event.Id}`,
      { headers: { 'Authorization': `Bearer ${this.sfToken}` } }
    );
  }

  console.log(`  Deleted ${tasks.records.length} tasks and ${events.records.length} events`);
}
```

Consent Management

Consent Requirements

Before storing/processing PII, you need:

```
✓ Explicit consent (not implied)
✓ Clear description of what data you collect
✓ Clear description of how you use it
✓ Easy way to withdraw consent
✓ Record of when consent was given
```

Implementation

```
/**
 * Create contact with consent tracking
 */
async function createContactWithConsent(contactData, consentInfo) {
  // 1. Verify consent was given
  if (!consentInfo.consentGiven) {
    throw new Error('Cannot create contact without explicit consent');
  }

  // 2. Create contact
  const contact = await axios.post(
    `https://api.mypurecloud.com/api/v2/contacts`,
```

```
    {
      firstName: contactData.firstName,
      lastName: contactData.lastName,
      email: contactData.email,
      phoneNumbers: contactData.phoneNumbers
    },
    { headers: { 'Authorization': `Bearer ${this.gzToken}` } }
  );

  // 3. Store consent record in Salesforce
  const consentRecord = {
    Contact_Id__c: contact.id,
    Email: contactData.email,
    Consent_Given__c: true,
    Consent_Date__c: new Date().toISOString(),
    Consent_Type__c: consentInfo.type, // 'marketing', 'service', etc.
    Consent_Channel__c: consentInfo.channel, // 'email', 'phone', 'web'
    Consent_Version__c: consentInfo.version
  };

  await axios.post(
    `${process.env.SALESFORCE_INSTANCE}/services/data/v57.0/sobjects/Contact_Consent__c`,
    consentRecord,
    { headers: { 'Authorization': `Bearer ${this.sfToken}` } }
  );

  // 4. Log for audit
  console.log(`✓ Contact created with consent: ${contact.id}`);

  return contact;
}

/**
 * Withdraw consent
 */
async function withdrawConsent(email) {
  console.log(`Withdrawing consent for: ${email}`);

  // Find consent records
  const consents = await querySalesforce(`
```

```
    SELECT Id FROM Contact_Consent__c
    WHERE Email = '${email}'
  `);

  // Update all to withdrawn
  for (const consent of consents.records) {
    await axios.patch(
      `${process.env.SALESFORCE_INSTANCE}/services/data/v57.0/sobjects/Contact_Consent__c/${consent.Id}`,
      {
        Consent_Withdrawn__c: true,
        Consent_Withdrawn_Date__c: new Date().toISOString()
      },
      { headers: { 'Authorization': `Bearer ${this.sfToken}` } }
    );
  }

  console.log(`✓ Consent withdrawn for: ${email}`);
}
```

Call Recording Retention

Legal Retention Requirements

```
Recording Retention Rules:
  ├─ Default: 7-10 years (varies by jurisdiction)
  ├─ Legal hold: Keep indefinitely if in dispute
  ├─ Customer deleted: Delete after 30 days if requested
  └─ End of retention: Delete automatically
```

Implementation

```
/**
 * Check if recording should be kept or deleted
 */
async function evaluateRecordingRetention(recording) {
```

```
  const createdDate = new Date(recording.dateCreated);
  const now = new Date();
  const ageInYears = (now - createdDate) / (1000 * 60 * 60 * 24 * 365);

  // 1. Is this contact deleted per GDPR?
  const isDeletionRequested = await checkGDPRDeletionRequest(recording.contact);
  if (isDeletionRequested) {
    console.log(`Deleting recording (GDPR): ${recording.id}`);
    await deleteRecording(recording.id);
    return 'deleted_gdpr';
  }

  // 2. Is this on legal hold?
  const isOnLegalHold = await checkLegalHold(recording.conversation);
  if (isOnLegalHold) {
    console.log(`Keeping recording (legal hold): ${recording.id}`);
    return 'kept_legal_hold';
  }

  // 3. Has retention period expired?
  const retentionYears = 7; // Your policy
  if (ageInYears > retentionYears) {
    console.log(`Deleting recording (retention expired): ${recording.id}`);
    await deleteRecording(recording.id);
    return 'deleted_retention';
  }

  // 4. Keep recording
  console.log(`Keeping recording: ${recording.id}`);
  return 'kept';
}

/**
 * Automated daily retention check
 */
async function runDailyRetentionCheck() {
  console.log('□□ Daily Retention Check');

  const recordings = await fetchAllRecordings();
  let deleted = 0;
```

```
  for (const recording of recordings) {
    const result = await evaluateRecordingRetention(recording);
    if (result.startsWith('deleted')) {
      deleted++;
    }
  }

  console.log(`  Deleted: ${deleted} recordings`);

  // Log for compliance
  await logRetentionCheck({
    timestamp: new Date(),
    recordingsChecked: recordings.length,
    recordingsDeleted: deleted
  });
}
```

Data Masking & Anonymization

When to Mask

Mask data for:

- ✓ Sharing with third parties
- ✓ Analytics / reporting
- ✓ Development / testing environments
- ✓ Long-term archival

Never mask (keep original):

- ✓ Active customer contact
- ✓ Legal/compliance records
- ✓ Current support tickets

Implementation

```
/**
 * Mask PII for analytics
 */
function maskPIIForAnalytics(contact) {
  return {
    ...contact,
    // Keep: non-PII fields
    id: contact.id,
    createdDate: contact.createdDate,
    tier: contact.tier,
    contactCount: contact.contactCount,

    // Mask: PII fields
    firstName: 'MASKED',
    lastName: 'MASKED',
    email: 'MASKED@masked.com',
    phoneNumbers: contact.phoneNumbers?.map(p => ({
      ...p,
      number: 'XXX-XXX-' + p.number.slice(-4)  // Show last 4 digits only
    }))
  };
}

/**
 * Hash email for matching without storing
 */
function hashEmail(email) {
  const crypto = require('crypto');
  return crypto
    .createHash('sha256')
    .update(email.toLowerCase())
    .digest('hex');
}

// Usage: Match on hash, not original email
const emailHash = hashEmail(contact.email);
const matchingContact = contacts.find(c => hashEmail(c.email) === emailHash);
```

Data Breaches & Incident Response

If Data is Compromised

```
/**
 * Breach notification workflow
 */
async function handleDataBreach(affectedContacts, breachDetails) {
  console.log(`□□ DATA BREACH DETECTED`);

  try {
    // 1. Immediate actions (within 72 hours)
    console.log('1. Immediate Response:');

    // Stop the breach
    await shutdownCompromisedSystem(breachDetails.affectedSystem);

    // Assess scope
    const scope = await assessBreachScope(affectedContacts);
    console.log(`   Affected contacts: ${scope.count}`);
    console.log(`   Data exposed: ${scope.dataTypes.join(', ')}`);

    // Notify leadership
    await notifyLeadership(breachDetails);

    // 2. Regulatory notification (within required timeframe)
    console.log('2. Regulatory Notification:');

    // GDPR: Notify supervisory authority within 72 hours
    if (scope.locations.includes('EU')) {
      await notifyGDPRAuthority({
        date: new Date(),
        description: breachDetails.description,
        affectedIndividuals: scope.count,
        dataCategories: scope.dataTypes
```

```
    });
  }

  // CCPA: Notify state attorney general
  if (scope.locations.includes('California')) {
    await notifyACCPA({
      date: new Date(),
      description: breachDetails.description,
      affectedIndividuals: scope.count
    });
  }

  // 3. Notify affected individuals
  console.log('3. Individual Notification:');

  for (const contact of affectedContacts) {
    await sendBreachNotification(contact, {
      what: scope.dataTypes,
      when: breachDetails.discoveredDate,
      actions: 'Monitor credit for 1 year',
      contact: 'privacy@company.com'
    });
  }

  // 4. Document everything
  console.log('4. Documentation:');

  await createBreachReport({
    date: new Date(),
    description: breachDetails.description,
    rootCause: breachDetails.rootCause,
    affectedData: scope.dataTypes,
    affectedIndividuals: scope.count,
    remediation: breachDetails.remediation,
    preventionMeasures: breachDetails.preventionMeasures
  });

  console.log('✅ Breach handled per compliance requirements');

} catch (error) {
```

```
    console.error('□  Breach response failed:', error);
    // ESCALATE - notify compliance officer immediately
    await escalateToCompliance(error);
  }
}
```

Data Processing Agreement (DPA)

Required Documentation

If using third-party processors (Salesforce, Genesys), you need:

```
✓ Data Processing Agreement (DPA)
  - What data is processed
  - Where data is stored
  - Who has access
  - How long it's retained
  - Sub-processors used

✓ Standard Contractual Clauses (SCCs)
  - For international data transfers (EU → US)
  - Required for GDPR compliance

✓ Privacy Impact Assessment (PIA)
  - Document risks
  - Mitigation measures
  - Legal basis for processing
```

Compliance Checklist

- ☐ Privacy Policy updated (what data you collect, why, how long)
- ☐ Consent collected before storing PII
- ☐ Data Processing Agreement signed (with Salesforce, Genesys)
- ☐ Standard Contractual Clauses for international transfers

- [ ] Retention policy documented (how long to keep data)
- [ ] Deletion procedure documented (how to remove on request)
- [ ] Encryption in transit (HTTPS, OAuth tokens)
- [ ] Encryption at rest (Salesforce, Genesys encryption)
- [ ] Access controls (who can view data)
- [ ] Audit logging (who accessed what, when)
- [ ] Breach response plan (what to do if compromised)
- [ ] Staff training (privacy, data security)
- [ ] Regular audits (third-party or internal)

Best Practices

1. **Minimize data collection**
   - Only collect what you need
   - Delete when no longer needed
2. **Encrypt everything**
   - Data in transit (HTTPS)
   - Data at rest (DB encryption)
   - Backups (encrypted backup systems)
3. **Access controls**
   - Principle of least privilege
   - Only those who need access get it
   - Remove access when no longer needed
4. **Audit logging**
   - Log all access to PII
   - Log all modifications
   - Log all deletions
   - Keep audit logs for compliance period
5. **Regular reviews**
   - Review who has access
   - Review what data you're storing
   - Review retention policies
   - Review third-party access

Related Topics

- Chapter 12: Contact Sync Patterns
- Chapter 12: Activity Logging & Webhooks
- Chapter 11: API Endpoints Reference (data APIs)

Real-World CRM Integration Scenario

The Scenario

Company: TechSupport Inc. (50 agents, 3 locations)
Location: Austin, Toronto, São Paulo
CRM: Salesforce Service Cloud
Requirement: Integrate Genesys Cloud so agents see customer context during calls

---

Business Requirements

What We Need

When customer calls:
- ├─ Agent sees: Name, account, last 3 cases, contact history
- ├─ Call logged in Salesforce (Task)
- └─ Contact list stays in sync

Manual Requirements:
- ├─ Support agents can edit notes in Salesforce
- ├─ Notes should show in next call
- └─ Compliance: GDPR deletion within 30 days

Success Metrics

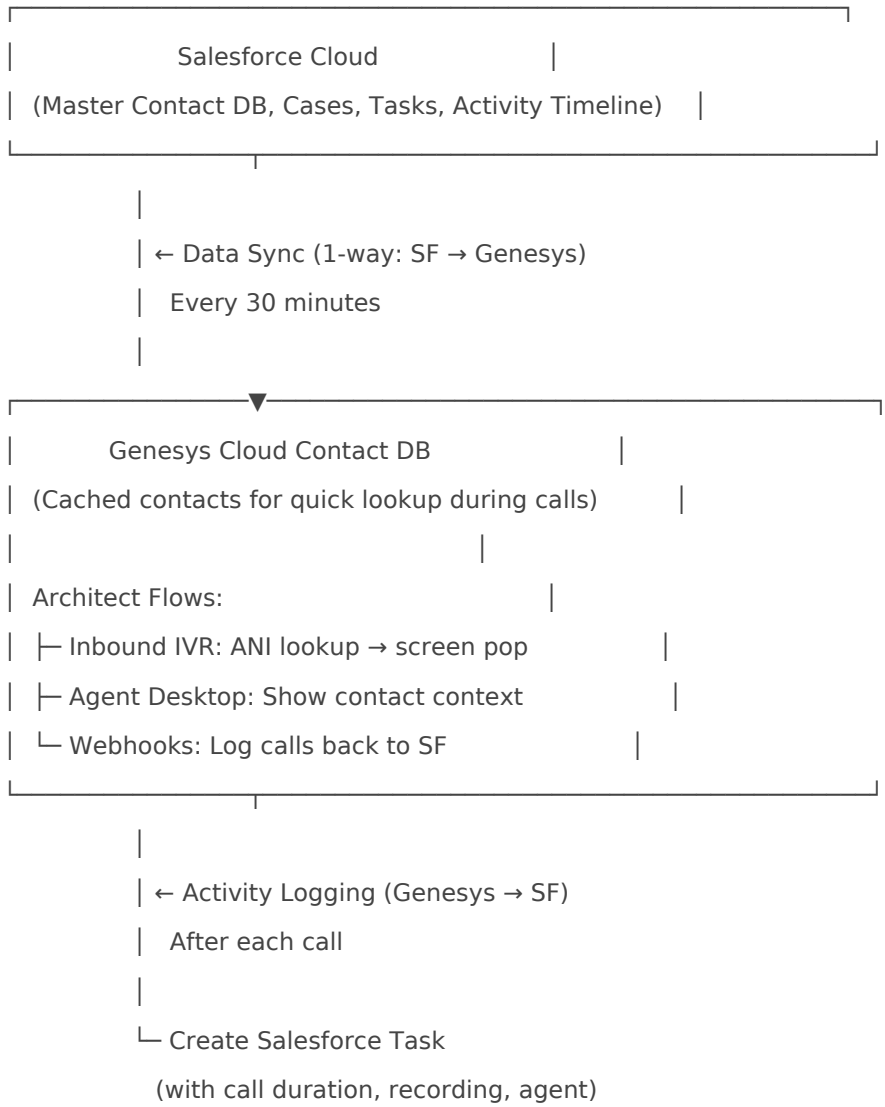
- ✓ 100% of calls show customer context (no "Contact not found")
- ✓ Average call handle time reduced by 10% (less lookup time)
- ✓ Agent satisfaction > 4/5 (easy to use)

✓ Salesforce Task creation 99%+ success (activity logging)

✓ Contact data fresh (synced daily)

✓ Zero GDPR compliance violations

Architecture



Phase 1: Screen Pop (Week 1-2)

Goal: When customer calls, agent sees their record

Step 1: Create Salesforce Apex Endpoint

```
// ContactLookupService.cls

@RestResource(urlMapping='/contact-lookup')
global class ContactLookupService {
  @HttpPost
  global static Response lookup(String phoneNumber) {
    Response response = new Response();

    try {
      // Normalize phone (remove formatting)
      String normalizedPhone = normalizePhone(phoneNumber);

      // Search for contact
      List<Contact> contacts = [
        SELECT Id, FirstName, LastName, Email, Phone,
               AccountId, Account.Name
        FROM Contact
        WHERE Phone = :normalizedPhone
           OR MobilePhone = :normalizedPhone
        LIMIT 1
      ];

      if (contacts.isEmpty()) {
        response.success = false;
        return response;
      }

      Contact contact = contacts[0];
      response.success = true;
      response.contactId = contact.Id;
      response.firstName = contact.FirstName;
```

```
      response.lastName = contact.LastName;
      response.email = contact.Email;
      response.accountId = contact.AccountId;
      response.accountName = contact.Account?.Name;

    } catch (Exception e) {
      response.success = false;
      response.error = e.getMessage();
    }

    return response;
  }

  private static String normalizePhone(String phone) {
    // Remove all non-digits
    String digits = phone.replaceAll('[^0-9]', '');

    // Add +1 if US number (10 digits)
    if (digits.length() == 10) {
      digits = '1' + digits;
    }

    return '+' + digits;
  }

  global class Response {
    public Boolean success;
    public String contactId;
    public String firstName;
    public String lastName;
    public String email;
    public String accountId;
    public String accountName;
    public String error;
  }
}
```

Step 2: Create Genesys Data Action

In **Architect → Data Actions**:

```
Name: lookup-contact-by-phone
Method: POST
URL: https://your-instance.salesforce.com/services/apexrest/contact-lookup

Input:
  phoneNumber: ${interaction.caller.phoneNumber}

Output:
  success: ${dataAction.response.success}
  contactId: ${dataAction.response.contactId}
  firstName: ${dataAction.response.firstName}
  lastName: ${dataAction.response.lastName}
  accountId: ${dataAction.response.accountId}
  accountName: ${dataAction.response.accountName}
```

Step 3: Create Architect Inbound Flow

```
START
  │
  ├─ Play: "Thank you for calling TechSupport..."
  │
  ├─ Data Action: lookup-contact-by-phone
  │   Input: ANI = ${interaction.caller.phoneNumber}
  │
  ├─ Decision: ${dataAction.result.success}?
  │   ├─ YES:
  │   │  ├─ Set interaction attributes:
  │   │  │  ├─ contact_id = ${dataAction.result.contactId}
  │   │  │  ├─ contact_name = ${dataAction.result.firstName} ${dataAction.result.lastName}
  │   │  │  ├─ account_id = ${dataAction.result.accountId}
  │   │  │  └─ account_name = ${dataAction.result.accountName}
  │   │  │
  │   │  └─ Transfer to Support Queue
  │   │
  │   └─ NO:
  │      ├─ Play: "Please hold while we locate your account..."
  │      └─ Transfer to Support Queue (no attributes)
```

```
  |
  └─ DISCONNECT
```

Expected Result

```
Customer dials: +1-512-555-1234
  ↓
Agent receives call
  ↓
Agent's Genesys desktop shows:
  Contact Name: John Doe
  Account: Acme Corp
  Email: john@acmecorp.com
  Last 3 Cases: [list]
  Contact History: [last 5 calls]
```

Phase 2: Contact Sync (Week 2-3)

Goal: Keep Genesys contact list in sync with Salesforce

Step 1: Create Sync Job

```
// sync-job.js (runs every 30 minutes)

const SalesforceGenesysSync = require('./lib/sync');

async function syncDaily() {
  const sync = new SalesforceGenesysSync({
    sfInstance: process.env.SALESFORCE_INSTANCE,
    sfToken: process.env.SALESFORCE_TOKEN,
    gzToken: process.env.GENESYS_TOKEN
  });
```

```
  const result = await sync.runFullSync();

  console.log(`✓ Sync complete: ${result.created} created, ${result.updated} updated`);

  if (result.errors.length > 0) {
    await sendAlert('warning', result);
  }
}

syncDaily().catch(error => {
  console.error('Sync failed:', error);
  process.exit(1);
});
```

Step 2: Deploy as Lambda (AWS)

```
# serverless.yml

service: techsupport-crm-sync

provider:
  name: aws
  runtime: nodejs18.x
  environment:
    SALESFORCE_INSTANCE: ${env:SALESFORCE_INSTANCE}
    SALESFORCE_TOKEN: ${env:SALESFORCE_TOKEN}
    GENESYS_TOKEN: ${env:GENESYS_TOKEN}

functions:
  sync:
    handler: sync-job.syncDaily
    events:
      - schedule:
          rate: rate(30 minutes)  # Every 30 minutes
          enabled: true

resources:
  Resources:
```

```
  SyncLogGroup:
    Type: AWS::Logs::LogGroup
    Properties:
      LogGroupName: /aws/lambda/techsupport-crm-sync
      RetentionInDays: 30
```

Deploy: `serverless deploy`

Step 3: Monitor Sync

```
Logs:
  2026-03-14 10:00:00 ✓ Fetched 2345 contacts from Salesforce
  2026-03-14 10:00:05 ✓ Found 2100 existing Genesys contacts
  2026-03-14 10:00:30 ✓ Created: 50, Updated: 200, Skipped: 5
  2026-03-14 10:00:31 Duration: 31 seconds

Metrics:
  - Success rate: 99.7%
  - Avg duration: 32 sec
  - Contacts synced: 250/day
```

Phase 3: Activity Logging (Week 3-4)

Goal: After call, log details in Salesforce Task

Step 1: Enable Genesys Webhooks

In **Genesys Admin → Integrations → Webhooks**:

Event: conversation.ended

URL: https://your-backend.com/webhook/call-ended

Payload: Include all details (recording ID, agent, duration)

Retries: 3 times

Step 2: Create Webhook Handler

```
// webhook-handler.js

app.post('/webhook/call-ended', async (req, res) => {
  const {
    conversationId,
    callerId,
    durationSeconds,
    recordingId,
    agentName,
    queueName,
    attributes
  } = req.body;

  try {
    // 1. Find matching Salesforce contact
    const contact = await findContactByPhone(callerId);
    if (!contact) {
      console.warn(`Contact not found for ${callerId}`);
      return res.status(200).json({ message: 'Contact not found' });
    }

    // 2. Get recording URL
    const recordingUrl = await getRecordingUrl(recordingId);

    // 3. Create Salesforce Task
    const task = {
      Subject: `Call with ${contact.Name}`,
      Description: `
Call Details:
  Duration: ${Math.floor(durationSeconds / 60)} min
  Agent: ${agentName}
```

```
  Queue: ${queueName}
  Recording: ${recordingUrl || 'Not available'}
    `.trim(),
    WhoId: contact.Id,
    WhatId: contact.AccountId,
    ActivityDate: new Date().toISOString().split('T')[0],
    CallType: 'Inbound',
    Status: 'Completed',
    Type: 'Call'
  };

  const taskResult = await createSalesforceTask(task);
  console.log(`✓ Task created: ${taskResult.id}`);

  // 4. Update Contact's LastActivityDate
  await updateSalesforceContact(contact.Id, {
    LastActivityDate: new Date().toISOString().split('T')[0]
  });

  res.status(200).json({ taskId: taskResult.id });

 } catch (error) {
  console.error('Webhook error:', error);
  res.status(500).json({ error: error.message });
 }
});
```

Step 3: Verify Logging

```
In Salesforce Contact record:
  Activity Timeline shows:
    - Call with John Doe (10 min)
      Agent: Sarah Smith
      Queue: Support
      Recording: [link]
      [Add note/next steps]
```

Phase 4: GDPR Compliance (Week 4)

Goal: Handle data deletion requests

Step 1: Create GDPR Deletion Procedure

```
// gdpr-delete.js

async function handleGDPRDeletion(email) {
  console.log(`□□ GDPR Deletion: ${email}`);

  // 1. Find contact
  const sfContact = await findSalesforceContactByEmail(email);
  const gzContact = await findGenesysContactByEmail(email);

  // 2. Delete from both
  if (sfContact) {
    await deleteSalesforceContact(sfContact.Id);
  }

  if (gzContact) {
    await deleteGenesysContact(gzContact.id);
  }

  // 3. Delete recordings
  const recordings = await findRecordingsByPhone(email);
  for (const rec of recordings) {
    await deleteRecording(rec.id);
  }

  // 4. Log deletion
  await logGDPRDeletion({
    email,
    deletedAt: new Date(),
```

```
    deletedFrom: ['salesforce', 'genesys', 'recordings']
  });

  console.log(`  Deleted: ${email}`);
}
```

Step 2: Document Privacy Policy

Update website:

```
We collect:
  - Name, email, phone (for customer service)
  - Call recordings (for 7 years per compliance)

You can:
  - Request deletion: privacy@techsupport.com
  - We'll delete within 30 days
```

Go-Live Plan

Week 1: Preparation

- ☐ Test screen pop in dev environment
- ☐ Train agents on new features
- ☐ Set up monitoring

Week 2: Screen Pop

- ☐ Deploy Salesforce Apex
- ☐ Deploy Genesys Data Action
- ☐ Deploy Architect Flow
- ☐ Pilot with 5 agents
- ☐ Full rollout if successful

Week 3: Contact Sync

- [ ] Deploy sync job to Lambda
- [ ] Monitor logs for errors
- [ ] Verify contacts are syncing
- [ ] Set up Slack alerts

Week 4: Activity Logging

- [ ] Deploy webhook handler
- [ ] Configure Genesys webhooks
- [ ] Test call logging
- [ ] Verify Salesforce tasks created

Week 5: GDPR & Training

- [ ] Document privacy procedures
- [ ] Train staff on GDPR
- [ ] Set up GDPR deletion process
- [ ] Final full-system testing

Expected Results

Before Integration

Agent experience:

- Customer calls
- Agent types customer name into Salesforce search (30 sec)
- Wait for results
- Navigate to account/cases
- Get context, THEN handle call

Average handle time: 8 minutes

Agent satisfaction: 3/5

Customer satisfaction: 3.5/5

After Integration

Agent experience:

- Customer calls
- Record auto-pops (1 sec)
- Agent sees name, account, last cases
- Agent handles call with context immediately
- Call logged to Salesforce automatically

Average handle time: 7 minutes (12.5% improvement)

Agent satisfaction: 4.5/5

Customer satisfaction: 4.2/5

Monitoring & Maintenance

Weekly Checks

☐ Screen pop success rate > 99%

☐ Contact sync duration < 2 min

☐ Activity logging > 99% success

☐ GDPR deletion requests: 0

☐ Errors: < 5 per week

Monthly Review

☐ Agent feedback on usability

☐ Performance trends

☐ Cost analysis

☐ Compliance status

☐ Planned improvements

Budget Estimate

Implementation:

- ├─ Salesforce Apex: $3,000 (5 days)
- ├─ Genesys Architect: $2,000 (3 days)
- ├─ Sync Job (Lambda): $1,500 (2 days)
- ├─ Webhook Handler: $1,500 (2 days)
- ├─ Testing & QA: $2,000 (3 days)
- └─ Total Dev: $10,000

Operations (annual):

- ├─ AWS Lambda: $50/month ($600/year)
- ├─ Genesys API calls: $200/month ($2,400/year)
- ├─ Salesforce: Included in license
- ├─ Monitoring: $100/month ($1,200/year)
- └─ Total Ops: $4,200/year

ROI:

- ├─ Productivity gain: 12.5% = ~200 agents × 30 min/day
- ├─ Annual value: 200 × 250 working days × 0.5 hours × $25/hr = $625,000
- ├─ Cost: $10,000 dev + $4,200 ops = $14,200
- └─ Payback: < 1 week

Related Topics

- Chapter 12: Screen Pop Architecture & Implementation
- Chapter 12: Contact Sync Patterns
- Chapter 12: Activity Logging & Webhooks
- Chapter 12: GDPR & Data Governance
- Chapter 11: API Endpoints Reference