Real-World Integration Patterns & Deployment

Common Integration Patterns

Pattern 1: Salesforce ↔ Genesys Contact Sync

Scenario: Synchronize Salesforce contacts to Genesys external contacts

Architecture:

Genesys Cloud API
↑
| PATCH /externalcontacts
| (partial updates, Mar 2026)
|
Sync Service
(Node.js/Python)
↑
| Query Salesforce API
|
Salesforce Org

Frequency: Every 30 minutes
Direction: Salesforce → Genesys (one-way)
Trigger: Scheduled job (cron)
Volume: ~1,000 contacts per sync

Authentication:

Salesforce:
├─ OAuth 2.0 (your existing setup)
├─ Service account or user credentials
└─ Connected app registered

Genesys:
├─ OAuth 2.0 Client Credentials
├─ Role: External Contact Manager
├─ Scopes: externalcontacts:manage
└─ Monthly secret rotation

Data Flow:

1. Query Salesforce:
   GET /services/data/v60.0/query
   SELECT Id, Email, Phone, Name, AccountId
   FROM Contact
   WHERE LastModifiedDate > :lastSync

2. Transform Data:
   ├─ Normalize phone (E.164)
   ├─ Validate email
   ├─ Map custom fields
   ├─ Add external ID (Salesforce ID)
   └─ Group by action (create/update)

3. Sync to Genesys:
   ├─ New contacts: POST /externalcontacts/contacts
   ├─ Updates: PATCH /externalcontacts/contacts/{id}
   ├─ Batches: Group every 50-100
   └─ Handle errors: Log and retry

4. Track Progress:
   ├─ Timestamp last successful sync
   ├─ Count created/updated/failed
   ├─ Send email notification
   └─ Log all activities

Implementation Highlights:

Error Handling:
├─ 409 Conflict: Already exists (update instead)
├─ 400 Bad Request: Invalid data (log and skip)
├─ 429 Rate Limited: Backoff and retry
├─ 503 Unavailable: Retry with backoff
└─ All other: Fail and alert

Partial Updates (PATCH):
├─ Update only changed fields
├─ Prevents overwriting unmanaged data
├─ Reduces payload size
├─ Better for CRM sync
└─ Available March 2026+

Idempotency:
├─ Use externalId for matching
├─ Prevent duplicate creates
├─ Retry-safe operations
├─ Track processed records
└─ Handle partial failures

Performance:
├─ Batch 50 contacts per request
├─ 1,000 contacts: ~20 requests
├─ Completes in < 5 minutes
├─ Well under rate limits
└─ Minimal API footprint

Benefits:
├─ Single source of truth
├─ Real-time contact availability
├─ No duplicate entry
├─ Reduces manual effort
└─ Improved agent experience

Pattern 2: Nightly Analytics Report Generation

Scenario: Generate daily contact center reports

Architecture:

Genesys Cloud Analytics API
↑
|
Report Generator Service
(scheduled, 2:00 AM daily)
↓
|
Email Distribution List

Authentication:

Client Credentials:
├─ Scopes: analytics:conversationDetail
├─ Token Duration: 1 hour (sufficient)
├─ No user interaction needed
└─ Fully automated

Workflow:

1. Authenticate (2:00 AM)
   POST /oauth/token
   grant_type: client_credentials
   Result: Access token

2. Query Analytics (2:01 AM)
   ├─ Yesterday's date range
   ├─ All queues/teams
   ├─ Conversations aggregate
   ├─ Service Level, AHT, ASA, Abandon Rate
   └─ Multiple requests (by dimension)

3. Generate Report (2:05 AM)
   ├─ Process data
   ├─ Create charts/graphs
   ├─ Format professionally
   ├─ Create PDF
   └─ Calculate YoY trends

4. Distribute (2:10 AM)
   ├─ Email PDF
   ├─ To stakeholders
   ├─ With summary
   └─ Archive for history

Report Contents:

Executive Summary:
├─ Total conversations: 5,432
├─ Service Level: 87% (Target: 80%)
├─ Avg Handle Time: 8:32 (Target: < 10min)
├─ Abandon Rate: 3.2% (Target: < 5%)
└─ Net Change vs yesterday: +2.1%

By Queue:
├─ Queue name
├─ Conversations
├─ Service Level
├─ AHT
├─ ASA
└─ Agents

Trending:
├─ Last 7 days
├─ Last 30 days
├─ YoY comparison
├─ Alerts (SLA failures)
└─ Recommendations

Implementation Highlights:

Scheduling:
├─ Cron: "0 2 * * *" (2:00 AM daily)
├─ Timezone: Your organization's TZ
├─ Alerting: If job fails
└─ Retry: Automatic

API Calls:
├─ /analytics/conversations/aggregates (multi-call)
├─ Total: 5-10 requests
├─ Completes in < 5 minutes
└─ No rate limit issues

Report Generation:
├─ Tool: ReportLab (Python) or similar
├─ Format: PDF
├─ Design: Professional template
└─ Data: Charts and tables

Email Distribution:
├─ Tool: SendGrid or SMTP
├─ Recipients: DL
├─ Schedule: Exactly 2:15 AM
├─ Archive: Save copy for history
└─ Tracking: Note delivery status

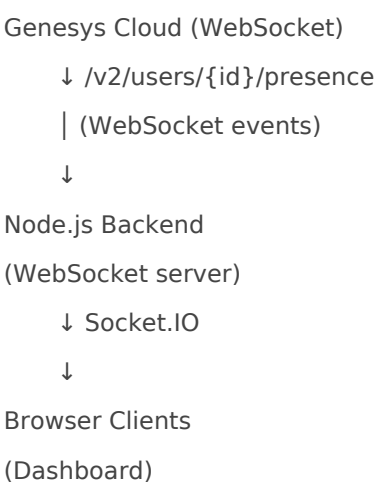
Error Handling:
├─ Authentication fails: Alert operations
├─ API call fails: Retry with backoff
├─ Report gen fails: Fallback to text
├─ Email fails: Retry next hour
└─ All failures: Log and notify

Benefits:
├─ Automated daily reporting
├─ Saves analyst 30 minutes/day
├─ Consistent delivery
├─ Stakeholders stay informed
└─ Data-driven decisions

Pattern 3: Real-Time Agent Status to Dashboard

Scenario: Display real-time agent availability in web dashboard

Architecture:



Authentication:

User Login:
├─ Authorization Code Grant
├─ User authenticates once
├─ Backend stores refresh_token
├─ Token includes necessary scopes
└─ presence:manage scope required

WebSocket Connection:
├─ Uses existing access_token
├─ Maintains connection
├─ Real-time events
└─ Automatic reconnection

Workflow:

1. User Logs In to Dashboard:
   ├─ Authorization Code flow
   ├─ User sees consent screen

```
   ├─ Backend gets access_token
   ├─ Backend stores refresh_token
   └─ User authenticated

2. Backend Establishes WebSocket:
   POST /api/v2/notifications/channels
   └─ Opens long-lived connection

3. Subscribe to Presence Events:
   PUT /api/v2/notifications/channels/{channelId}/subscriptions

   Subscriptions:
   ├─ v2.users.{user1}.presence
   ├─ v2.users.{user2}.presence
   ├─ v2.users.{user3}.presence
   └─ For all agents

4. Receive Real-Time Events:
   {
    "eventBody": {
      "userId": "user-123",
      "presenceDefinition": {
        "systemPresence": "available",
        "customPresences": [...]
      }
    }
   }

5. Update Dashboard:
   ├─ Forward event to browsers (Socket.IO)
   ├─ Update agent status display
   ├─ Change color/icon
   ├─ Show "Available", "Break", "Busy", etc
   └─ Real-time synchronization

Implementation Highlights:

Subscription Efficiency:
├─ Single WebSocket connection
├─ Multiple subscriptions
```

├─ vs: 100 polling requests/minute
├─ 99% reduction in API calls
└─ Real-time delivery

Connection Management:
├─ Maintain connection
├─ Automatic reconnection on failure
├─ Health checks
├─ Graceful degradation
└─ Error handling

Scalability:
├─ 1 backend: 100+ agent subscriptions
├─ vs: Polling → rate limited
├─ WebSocket: Event-driven
├─ Lower CPU usage
└─ Lower bandwidth

Error Recovery:
├─ Connection drops: Auto-reconnect
├─ Subscription fails: Retry
├─ Event deserialization: Log and skip
├─ Token expires: Refresh and reconnect
└─ Graceful fallback: Polling backup

Benefits:
├─ Real-time agent status
├─ Sub-second updates
├─ No polling overhead
├─ Better user experience
├─ Reduced infrastructure load
└─ Scalable solution

Deployment Strategies

Strategy 1: Development Environment

Setup:

OAuth Client:
├─ Grant Type: Authorization Code + PKCE
├─ Redirect URI: http://localhost:3000/callback
├─ Scopes: conversations:readonly
├─ Token Duration: 3600 (1 hour)
└─ Created by: Development team

Client Credentials (if needed):
├─ Grant Type: Client Credentials
├─ Scopes: externalcontacts:manage
├─ Token Duration: 3600
└─ Role: External Contact Manager (dev only)

Secrets Storage:
├─ .env file (local development)
├─ .gitignore entries:
│ ├─ .env
│ ├─ .env.local
│ └─ credentials/
├─ Example .env:
│ ├─ GENESYS_CLIENT_ID=dev-client-id
│ ├─ GENESYS_CLIENT_SECRET=dev-secret
│ └─ GENESYS_REGION=mypurecloud.com
└─ Never commit secrets!

Configuration:
├─ Use environment variables
├─ Different per developer
├─ Allow local overrides
├─ Development region specified
└─ Non-production data only

Testing:
├─ Unit tests: Mock API responses
├─ Integration tests: Dev Genesys org
├─ Manual testing: Full flow
├─ Error scenario testing

└ Rate limit testing

CI/CD Pipeline:
├ Run on: Developer machine
├ Linting: Yes
├ Unit tests: Yes
├ Build: Yes
├ Deploy: Local only
└ Secrets: Via .env (not checked in)

Monitoring:
├ Logging: Console output
├ Debugging: Browser DevTools
├ API calls: Inspect requests
├ Errors: Stack traces
└ Performance: Basic measurements

Strategy 2: Staging Environment

Setup:

OAuth Clients:
├ Separate from production
├ Authorization Code client
├ Client Credentials client
├ Different secrets
└ Same Genesys region (or test region)

Secrets Storage:
├ Environment variables (CI/CD platform)
├ Encrypted vault
├ GitHub Actions secrets / GitLab CI variables
├ Rotate monthly
└ Different from production

Configuration:
├ Staging-specific settings
├ Same region as production

├─ Test data only
├─ Verbose logging enabled
└─ Performance testing configured

Testing:
├─ Full integration tests
├─ End-to-end workflows
├─ Load testing (small scale)
├─ Error scenario testing
├─ Performance baselines
└─ Compatibility testing

CI/CD Pipeline:
├─ Trigger: On PR/Merge to staging branch
├─ Linting: Yes
├─ Unit tests: Yes
├─ Integration tests: Yes
├─ Build: Yes
├─ Deploy: Automated
├─ Smoke tests: Post-deploy
└─ Notify: On success/failure

Monitoring:
├─ Application logs: Aggregated
├─ Error tracking: Sentry/similar
├─ Performance monitoring: APM
├─ Uptime monitoring: Synthetic
└─ Alerts: To development team

Approval Process:
├─ Code review: Required
├─ Tests: Must pass
├─ Deployment: Semi-automated
├─ Rollback: Available
└─ Notify stakeholders

Data Management:
├─ Test data only
├─ Reset daily/weekly

├─ Production data: Never
├─ GDPR compliant
└─ Privacy respected

Strategy 3: Production Environment

Setup:

OAuth Clients:
├─ Separate production clients
├─ Multiple clients: Web app, services, etc.
├─ Different secrets per environment
├─ Rotate monthly minimum
├─ Monthly rotation documented
└─ Secure secret storage required

Secrets Storage:

Critical - Use Secure Vault:
├─ HashiCorp Vault (recommended)
├─ AWS Secrets Manager
├─ Azure Key Vault
├─ Google Cloud Secrets
└─ Store & access policies:
  ├─ Encrypt at rest
  ├─ Encrypt in transit
  ├─ Audit all accesses
  ├─ RBAC (role-based)
  ├─ Rotation tracking
  └─ Alert on unauthorized access

Configuration:

Environment-Specific:
├─ Production region only
├─ Performance settings optimized
├─ Logging: Moderate (balance vs storage)
├─ Monitoring: Comprehensive
├─ Alerting: All channels

└─ Recovery procedures: Documented

CI/CD Pipeline:

Strict Requirements:
├─ All tests: Must pass
├─ Code review: Mandatory
├─ Approval: Required (2+ reviewers)
├─ Build: Automated & verified
├─ Deploy: Gated release
├─ Smoke tests: Post-deploy (critical)
├─ Rollback: Instant available
└─ Notify: Multiple teams

Deployment Procedure:

1. Code Merge (to main branch)
   ├─ Tests pass
   ├─ Reviews approved
   └─ CI/CD starts

2. Build (automated)
   ├─ Code compiled
   ├─ Tests run
   ├─ Artifact created
   └─ Ready for deploy

3. Stage (automated)
   ├─ Deploy to staging
   ├─ Smoke tests run
   ├─ If all pass: await approval
   └─ If any fail: block deployment

4. Approve (manual gate)
   ├─ Engineering manager: approval
   ├─ On-call engineer: ready
   ├─ Time window: Business hours preferred
   └─ Cancellation: Available anytime

5. Deploy (automated)

├─ Deploy to production
├─ Gradual rollout (optional)
├─ Monitor deployment
├─ Health checks pass
└─ Notify team

6. Monitor (continuous)
├─ Watch logs
├─ Watch metrics
├─ Watch errors
├─ Alert threshold: Low
└─ Quick response ready

7. Rollback (if needed)
├─ Detected: Issue identified
├─ Decision: Rollback authorized
├─ Execute: One command
├─ Verify: Health checks pass
└─ Notify: Team aware

Monitoring:

Comprehensive Observability:
├─ Application logging (ELK/Splunk)
├─ Error tracking (Sentry/Rollbar)
├─ APM (New Relic/DataDog)
├─ Uptime monitoring (Synthetic)
├─ Custom metrics
├─ Infrastructure monitoring
└─ Security monitoring

Alerting:

Critical (Immediate - Page):
├─ Application down (503/504)
├─ Authentication failures spike
├─ Database connection errors
├─ Memory/CPU critically high
├─ Deployment failed
└─ Security incident

High Priority (30 min - Slack):
├─ Error rate > 5%
├─ Response time spike
├─ Rate limit approached
├─ Token refresh failures
├─ API quota exceeded
└─ Unusual traffic pattern

Medium Priority (1 hour - Email):
├─ Minor errors
├─ Performance degradation
├─ Deprecated API usage
├─ Scheduled job delayed
└─ Configuration drift

Disaster Recovery:

Backup & Recovery:
├─ Database: Daily snapshots
├─ Configuration: Version controlled
├─ Secrets: Vault with audit logs
├─ RTO: < 1 hour
├─ RPO: < 15 minutes
└─ Tested monthly

Communication:

During Incident:
├─ Status page: Updated immediately
├─ Slack: Team notification
├─ Email: If long duration
├─ Customers: If customer-facing
└─ Escalation: Clear path

Post-Incident:
├─ Post-mortem: Scheduled
├─ Root cause: Identified
├─ Fix: Implemented
├─ Prevention: For future

└─ Lessons learned: Documented

Compliance:

Security Requirements:
├─ PCI-DSS: For payment data
├─ HIPAA: For health data
├─ SOC 2: For SaaS
├─ GDPR: For EU data
└─ Industry-specific: As applicable

Audit & Compliance:
├─ Audit logs: Retained 2+ years
├─ Access logs: Who did what when
├─ Change logs: All changes recorded
├─ Compliance checks: Quarterly
└─ External audits: Annual

Troubleshooting Deployments

Common Issues:

Problem: Authentication Fails in Production

Causes:
├─ Secret rotated, app not updated
├─ Secret expired (if revoked)
├─ Client deleted
├─ Secret wrong (typo)
└─ Wrong client for environment

Diagnosis:
├─ Check error: Invalid credentials
├─ Verify secret matches what stored
├─ Verify client still exists
├─ Check expiration date
└─ Try authentication manually

Fix:
├─ If secret wrong: Update immediately
├─ If client deleted: Recreate
├─ If expired: Generate new secret
├─ If revoked: Verify it's revoked
└─ Test after fix

Prevention:
├─ Document secret location
├─ Rotate on schedule
├─ Test auth before deploy
├─ Use vault for storage
└─ Alert before expiration

Problem: Rate Limiting in Production

Symptoms:
├─ 429 errors increasing
├─ Request latency increasing
├─ API calls slowing down
├─ Partial failures
└─ Customers reporting issues

Diagnosis:
├─ Check request volume
├─ Check request frequency
├─ Identify hot endpoints
├─ Check X-Rate-Limit-Remaining
└─ Calculate current rate

Fix (Short-term):
├─ Implement backoff
├─ Reduce request frequency
├─ Queue requests
├─ Reduce batch size
└─ Wait for window reset

Fix (Long-term):
├─ Use bulk endpoints

├─ Implement caching
├─ Use WebSocket events
├─ Optimize queries
└─ Consider enterprise limits

Prevention:
├─ Load test before production
├─ Monitor rate limit headers
├─ Alert when approaching limit
├─ Design for pagination
└─ Use bulk APIs from start

Problem: Token Refresh Failing

Symptoms:
├─ API calls return 401
├─ Refresh token errors
├─ Users getting logged out
├─ Authentication failing
└─ Errors in logs

Diagnosis:
├─ Check refresh token expired
├─ Check client exists
├─ Check secret correct
├─ Check token lifetime
└─ Try manual refresh

Fix:
├─ If expired: User must re-authenticate
├─ If client wrong: Update config
├─ If secret wrong: Update secret
├─ If lifetime: Adjust config
└─ Test refresh manually

Prevention:
├─ Refresh proactively (5 min before)
├─ Monitor refresh success rate
├─ Alert on refresh failures
├─ Document refresh logic

└ Test token refresh

Problem: Data Loss / Sync Issues

Symptoms:
├ Contacts not syncing
├ Data inconsistencies
├ Partial updates missing
├ Database conflicts
└ Customers reporting issues

Diagnosis:
├ Check sync logs
├ Verify API calls succeeded
├ Check for conflict errors
├ Verify data format
└ Compare source vs target

Fix:
├ Re-run sync
├ Correct data format
├ Handle conflicts
├ Verify completeness
└ Reconcile manually if needed

Prevention:
├ Implement idempotency
├ Use external IDs
├ Log all changes
├ Verify sync completion
├ Regular reconciliation
├ Alerts on failures
└ Testing with real data

Problem: Performance Degradation

Symptoms:
├ Slow API response times
├ High latency (p99)
├ User complaints

├─ Dashboard sluggish
└─ Timeout errors

Diagnosis:
├─ Check API latency
├─ Check application latency
├─ Check database latency
├─ Check network latency
└─ Identify bottleneck

Fix (Short-term):
├─ Scale up application
├─ Scale up database
├─ Clear cache
├─ Reduce requests
└─ Optimize queries

Fix (Long-term):
├─ Use caching layer
├─ Optimize database
├─ Use CDN
├─ Async processing
└─ Horizontal scaling

Prevention:
├─ Load test regularly
├─ Monitor latency
├─ Alert on degradation
├─ Capacity planning
└─ Performance budget

Key Takeaways: Chapter 8

- **Pattern Diversity** - Different patterns for different scenarios
- **Authentication Clear** - Always OAuth 2.0 (Client Credentials or Code)
- **Scheduling Critical** - Cron jobs for reliable automation
- **Error Handling** - Implement retry logic with backoff
- **Data Quality** - Validate and normalize before syncing

- **Deployment Gates** - Approval required for production
- **Monitoring Essential** - Know when things break
- **Documentation Important** - For debugging and maintenance

---

Interview Prep: Integration Patterns

Question	Answer
Salesforce sync pattern?	Query Salesforce, transform, batch POST/PATCH to Genesys
Report generation?	Scheduled job (cron), query analytics, generate PDF, email
Real-time status?	WebSocket subscriptions, event-driven updates, low overhead
Authentication type?	Client Credentials for services, Auth Code for users
Backoff strategy?	3, 9, 27 seconds, then 5-min increments
Error handling?	Retry on 429/5XX, fail on 4XX (except 401)
Data sync quality?	Validate, normalize, idempotent, external IDs
Deployment gate?	Approval required, smoke tests pass, monitoring ready
Secret storage?	Secure vault (Hashicorp, AWS, Azure)
Monitoring?	Logs, metrics, errors, alerts (critical/high/medium)

Document Version

Chapter: 8 of 8
Last Updated: March 2026
Status: Current with OAuth 2.0 & API standards
Scope: Integration patterns, deployment strategies, troubleshooting

---

Revision #1
Created 14 March 2026 19:33:54 by Cesar Gzz
Updated 14 March 2026 19:34:07 by Cesar Gzz