Case-Third-Party-Reader Technical Report

Executive Summary

The Case-Third-Party-Reader is a critical bidirectional synchronization service in Datadog's case management domain that enables seamless integration between Case Management and third-party ticketing systems. It consumes events from collaboration integration Kafka topics (Jira and ServiceNow webhooks) and applies validated changes to the internal Case Management system. The service implements sophisticated filtering logic to prevent synchronization loops, respects project-specific configuration, and handles custom field mappings while maintaining data consistency through optimistic locking.

1 Service Overview

1.1 Primary Functions

1. **Kafka Event Consumption**: Consumes events from separate Kafka topics for Jira and ServiceNow integrations
2. **Event Decoding**: Transforms third-party webhook payloads into normalized changelog structures
3. **Intelligent Filtering**: Two-stage filtering (PreFilter and Filter) to optimize performance and prevent sync loops
4. **State Synchronization**: Applies validated changes to cases including title, description, priority, status, assignee, due dates, and comments
5. **Custom Field Support**: Handles custom field mappings, particularly for Jira custom due date fields
6. **Conflict Detection**: Validates state consistency before applying changes to prevent concurrent update conflicts

1.2 Supported Integrations

Jira

- Event types: `jira:issue_updated`, `comment_created`
- Bidirectional sync for all standard fields and custom due date fields
- Metadata caching with 5-minute TTL for performance optimization
- State conflict detection for priority, status, and assignee changes

ServiceNow

- Event types: Comments (including work notes) and status changes
- Instance-based filtering with configuration caching
- Sync cycle prevention through author detection

1.3 Key Metrics

- **Deployment Flavors**: 2 separate deployments (Jira, ServiceNow)
- **Consumer Group**: case-management (shared with other case management consumers)
- **Replicas**: 1 per flavor deployment
- **Filter Reasons**: 27 distinct filtering reasons for comprehensive observability
- **Health Checks**: Liveness and readiness probes on port 8080
- **Deployment Environments**: Staging, Production, Government (FIPS-compliant)

2 Architecture Overview

2.1 System Architecture

The system follows a flavor-based deployment architecture where separate service instances handle different third-party integrations. Each flavor consumes from its dedicated Kafka topic and processes events according to integration-specific logic while sharing common processing patterns through abstraction interfaces.

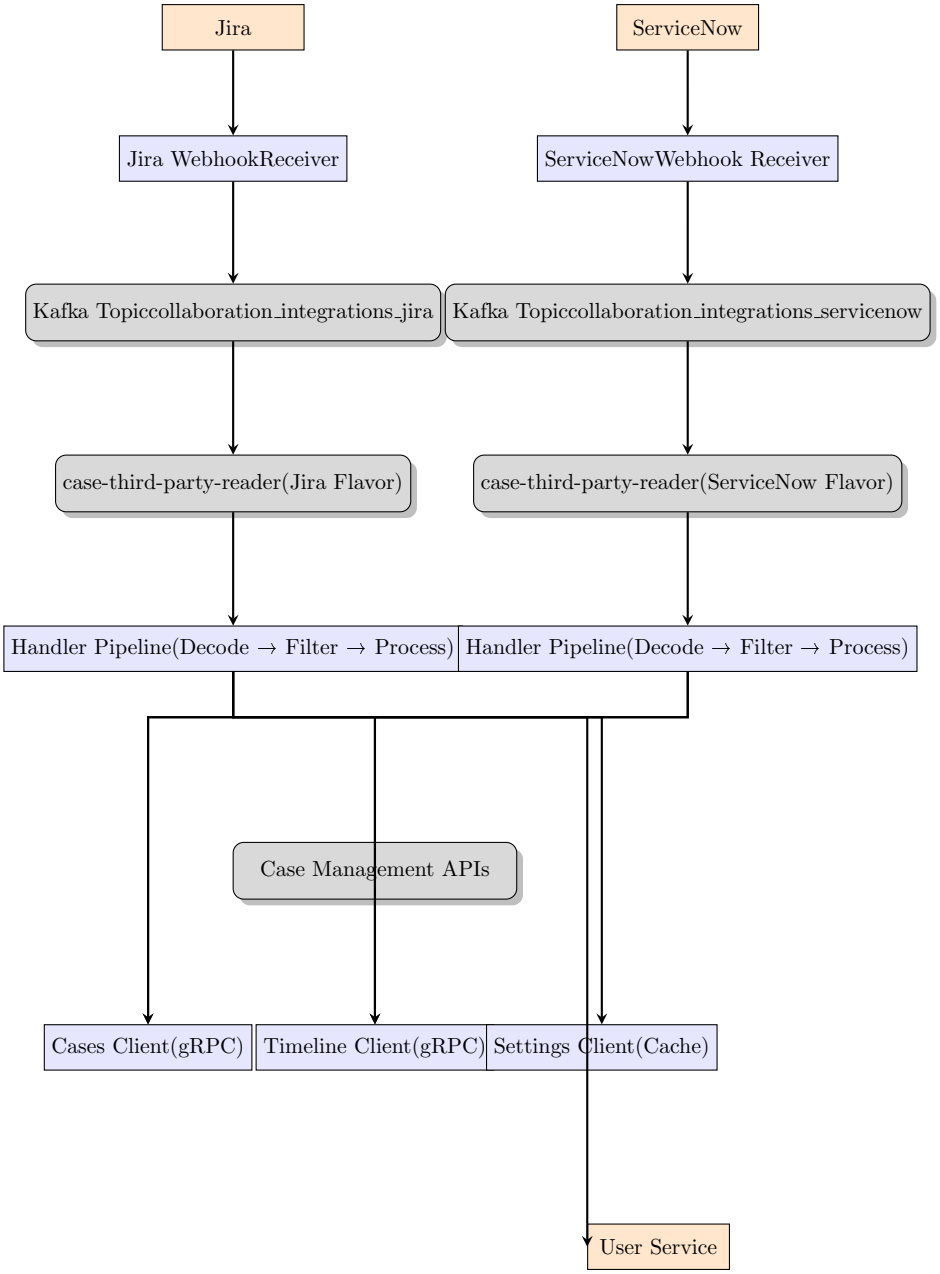

Figure 1: System Architecture

2.2 Event Flow Architecture

The event flow demonstrates the end-to-end journey from third-party systems to Case Management, highlighting the two-stage filtering process that optimizes performance by deferring expensive operations until necessary.

Event Processing Flow

3 Event Platform Integration

3.1 Kafka Consumer Configuration

- **Consumer Group**: case-management
- **Topics**:
  - `collaboration_integrations_jira` (Jira flavor)
  - `collaboration_integrations_servicenow` (ServiceNow flavor)
- **Offset Management**: Manual with auto-commit (5s interval)
- **Auto Offset Reset**: latest
- **Partition Strategy**: cooperative-sticky
- **Message Encoding**: JSON payloads with nested third-party webhook data
- **Acknowledgment Mode**: Explicit acknowledgment after successful processing

3.2 Event Processing Pipeline

The event processing pipeline implements a sophisticated multi-stage approach:

1. **Message Decoding**: Parse JSON payload from Kafka message into Go structs
2. **Changelog Extraction**: Transform third-party webhook data into normalized changelog structure
3. **PreFilter Validation**:
   - Synchronization cycle detection (events from Datadog are skipped)
   - Metadata validation against cached project configurations
   - Custom field evaluation for Jira due date fields
   - No external service calls for optimal performance
4. **Case and Settings Lookup**: Fetch case data and project settings from cache/gRPC
5. **Filter Validation**:
   - Bidirectional sync enablement checks
   - Field-level mapping validation (priority, status values)
   - State conflict detection using changelog `from` values
   - Sync time window validation (created_before_start_syncing_from)
6. **Changelog Processing**: Apply changes via Case Management APIs with optimistic locking
7. **Error Handling**:
   - Retryable errors: Pod crashes for clean state retry
   - Non-retryable errors: Event dropped with metrics
   - State conflicts: Filtered out to prevent overwriting concurrent updates

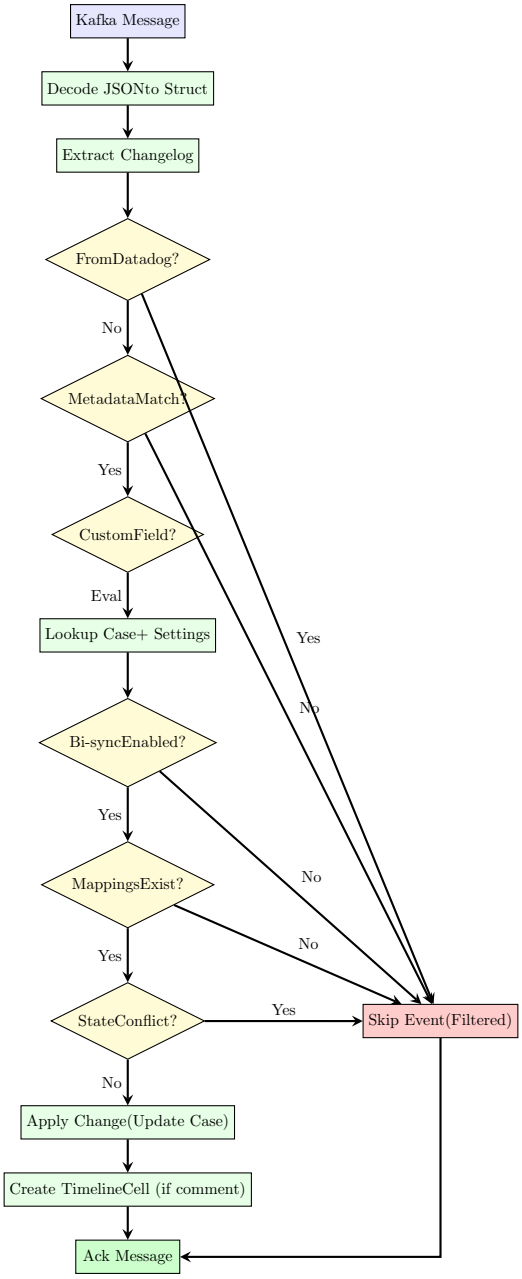

Figure 2: Detailed Event Processing Pipeline

4 Integration Details

4.1 Jira Integration

Event Types and Supported Fields

- **jira:issue_updated**
  - Title (summary) - Text field
  - Description - Markdown/rich text field
  - Priority - Mapped via Jira priority ID → Case priority name
  - Status - Mapped via Jira status ID → Case status name
  - Assignee - Resolved via display name → Datadog user UUID

- Due Date - Standard `duedate` field or custom fields (e.g., `customfield_10015`)

- **comment_created**
  - Comment body synced to Case timeline
  - Author information preserved

Custom Field Handling

Jira supports extensive custom fields beyond the standard schema. The service handles custom due date fields through configuration-driven evaluation:

1. Custom fields initially appear as `unsupported_{field_id}` in webhook payloads
2. During PreFilter, the service checks project settings for `jiraFieldId` configuration
3. If a custom field (e.g., `customfield_10015`) is configured as a due date field, the changelog type is transformed to `update_due_date`
4. The processor extracts the due date value from the `toID` field (ISO 8601 format)

Metadata Caching

To optimize PreFilter performance, the service maintains a TTL cache (5 minutes) of used Jira metadata per organization:

- Cache key: (jiraAccountID, projectID, issueTypeID)
- Benefit: Avoids expensive settings lookup for events from unconfigured Jira projects
- Tradeoff: Up to 5 minutes delay for new project configurations to take effect

State Conflict Detection

For priority, status, and assignee changes, the service implements state conflict detection:

1. Jira webhook includes `from` and `to` values in changelog
2. Service fetches current case state
3. If case's current value does not match changelog's `from` value, the change is filtered out
4. Prevents overwriting concurrent updates from other sources (e.g., Datadog UI)
5. Filtered with reason: `state_conflict`

4.2 ServiceNow Integration

Event Structure

ServiceNow events follow a diff-based structure:

```
{
  "type": "datadog_x_incident",
  "orgId": 123456,
  "sysId": "abc123...",
  "instanceName": "company",
  "updatedTime": "2024-01-15T10:30:00Z",
  "updatedBy": "john.doe",
  "data": {
    "comment": "Issue resolved",
    "state": "7",
    "correlationId": "case-uuid",
    ...
  }
}
```

Supported Operations

- **Comments**: Both standard comments and work notes are synced to Case timeline
- **Status Changes**: ServiceNow state field mapped to Case status names via settings
- **Sync Types**:
  - ST_BIDIRECTIONAL: Full two-way sync
  - ST_ONE_WAY_PULL: ServiceNow → Case Management only

Instance Caching
Similar to Jira metadata caching, ServiceNow uses instance-based caching:

- Cache key: (orgID, instanceName)
- TTL: 5 minutes
- Purpose: Skip events from unconfigured ServiceNow instances during PreFilter

Sync Cycle Prevention
ServiceNow events include an `updatedBy` field. Events where `updatedBy` equals "datadog" (case-insensitive) are filtered out with reason `break_synchronization_cycle`.

5 Datacenter Deployment Strategy

5.1 Multi-Environment Deployment

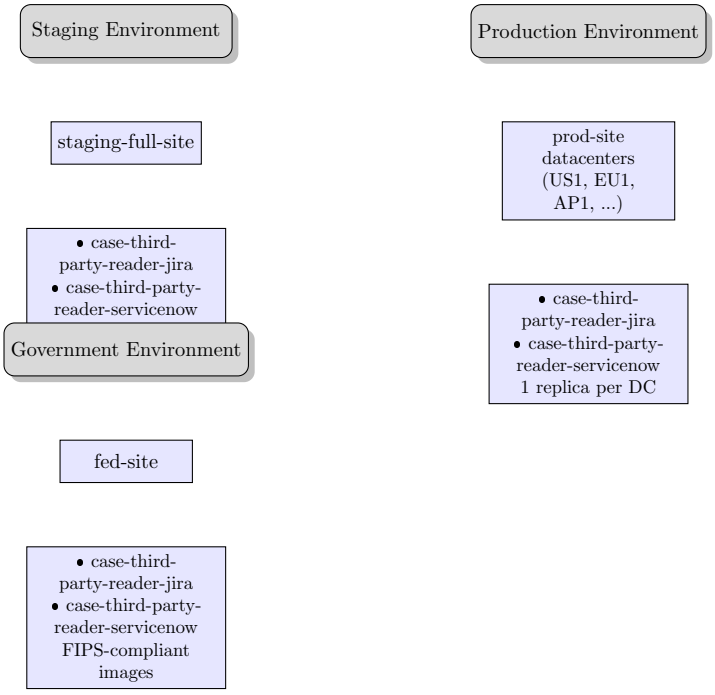

Figure 3: Deployment Strategy Across Environments

5.2 Deployment Configuration

Table 1: Deployment Settings by Environment

Setting	Staging	Production	Government
Datacenters	staging-full-site	prod-site (multiple)	fed-site
Replicas/Flavor	1	1 per DC	1
Deployment Strategy	dangerous_everything_parallel	delta (rolling)	delta (rolling)
Image	casethirdpartyreader	casethirdpartyreader	casethirdpartyreaderfips
Schedule	Mon 5:00 AM	Mon 12:30 PM	Wed 12:30 PM
Health Gates	Basic monitoring	Business hours + monitors	Business hours + monitors

5.3 Rolling Update Strategy

- **Max Unavailable**: 0 (zero-downtime deployments)
- **Max Surge**: 1 (one extra pod during updates)
- **Termination Grace Period**: Configurable via values (default allows graceful Kafka consumer shutdown)
- **Health Check Initial Delay**: 30 seconds for liveness and readiness
- **Monitor Gate**: Requires 20 minutes (1200s) of healthy metrics before progressing

6 Performance and Observability

6.1 Key Metrics Collected

Latency Metrics

```
case.case_third_party_reader.end_to_end_latency (distribution)
Tags: org_id, changelog_type, flavor,
      jira_account_id, jira_project_id
Description: Time from third-party event timestamp
             to processing completion
```

Processing Metrics

```
dd.case_third_party_reader.changelog_processed (count)
Tags: org_id, flavor, changelog_type
Description: Successfully processed changelogs

dd.case_third_party_reader.jira.filtered (count)
Tags: org_id, flavor, changelog_type, reason
Description: Events filtered with specific reason
             (27 distinct reasons)

dd.case_third_party_reader.decoding_failed (count)
Tags: flavor, error
Description: Failed to decode Kafka payload

dd.case_third_party_reader.retries (count)
Description: Pod crashed due to retryable error

dd.case_third_party_reader.out_of_order_event (count)
Tags: integration, org_id
```

```
Description: Event received out of chronological order
```

Kafka Consumer Metrics

```
dd.case.kafka_consumer.decode_error
dd.case.kafka_consumer.message_skipped
```

6.2 Filter Reasons for Observability

The service tracks 27 distinct filter reasons, enabling precise debugging and monitoring:

- break_synchronization_cycle
- field_jira_bi_sync_disabled
- field_value_not_mapped
- state_conflict
- no_corresponding_case
- project_not_found
- created_before_start_syncing_from
- integration_disabled
- project_sync_disabled
- unsupported_changelog_type
- no_matching_metadata
- custom_field_not_configured
- assignee_resolution_failed
- case_version_mismatch

...and 13 more

6.3 Dashboards and Monitoring

Datadog Dashboards

- **Production**: `https://app.datadoghq.com/dashboard/uc8-w7v-fca`
- **Staging**: `https://ddstaging.datadoghq.com/dashboard/42h-bsz-bmd`

Dashboard panels include:

- Event processing throughput by flavor and changelog type
- End-to-end latency percentiles (p50, p95, p99)
- Filter reason breakdown (pie chart)
- Error rates and retry counts
- Out-of-order event frequency
- Kafka consumer lag

Health Gates

```
monitorGate:
  query: service:case-third-party-reader
         AND team:case-management
  isHealthyTimeSeconds: 1200  # 20 minutes
  scopes: [DATACENTER]

businessHoursGate:
  preset: US_AND_EU
```

6.4 Logging and Tracing

Structured Logging

- **Format**: dd-go-std (JSON structured logs)
- **Data Access Control**: `dac.dataset: case_management_third_party_syncing_logs`
- **Restricted Access**: Case Management engineers only
- **Excluded Fields**: Long text fields (comments, descriptions) to prevent processing failures
- **Key Fields**: `changelog_type`, `third_party_type`, `org_id`, `evt_timestamp`, `author`, `is_from_datadog`
- **Jira-Specific**: `jira_issue_key` extracted from `third_party_id`

Distributed Tracing
APM spans hierarchy:

```
handler_{flavor}                  # Root span
          decode_event
          prefilter_third_party_changelog
          filter_third_party_changelog
                get_case
                get_settings
          process_changelog
              update_case
              create_timeline_cell
```

7 Error Handling Strategy

7.1 Error Classification

The service implements a three-tier error handling strategy:

1. **Skip Errors** (Non-Blocking)
   - Filtered events (all 27 filter reasons)
   - Invalid payloads (codes.InvalidArgument)
   - Decoding failures
   - Action: Log, record metric, acknowledge message
2. **Retry Errors** (Transient)
   - State conflicts (codes.Aborted)
   - Retriable API failures
   - Temporary service unavailability
   - Action: Exponential backoff via retrier library
   - Fallback: Pod crash after max retries (Kubernetes restarts with clean state)
3. **Permanent Errors** (Discard)
   - Malformed events that pass decoding but fail validation
   - Unsupported event types
   - Action: Log error, record metric, acknowledge message (prevent poison messages)

7.2 Crash-on-Retry Policy

After exhausting retries (exponential backoff with jitter), the service intentionally crashes:

Rationale:

- Kubernetes restarts the pod with a clean state
- Kafka consumer offset is not committed, so messages are reprocessed
- Avoids stuck offsets that block partition processing
- Metric: `dd.case_third_party_reader.retries`

Prevents:

- Kafka partition starvation
- Memory leaks from accumulated retry state
- Consumer group rebalancing issues

7.3 Operational Safety: Consul Message Skipping

For emergency situations (e.g., poison message blocking partition), operators can configure message skipping via Consul:

```
[dd.kafka.consumer]
skip_messages = offset1,offset2,...
```

Skipped messages are:

- Logged with full details
- Counted in `dd.case.kafka_consumer.message_skipped`
- Acknowledged to unblock partition

8 Security and Compliance

8.1 Security Features

- **Service Account**: Dedicated Kubernetes service account per deployment with RBAC
- **Network Policies**: Cilium network policies for service isolation
  - No explicit ingress rules (service does not accept inbound HTTP traffic)
  - Egress managed by Fabric defaults
- **TLS**: All gRPC communications to Case Management APIs use TLS encryption
- **Authentication**: OUI client for user service authentication
- **Data Access Control**: Logs restricted to Case Management team

8.2 FIPS Compliance

Government environment deployments use FIPS 140-2 compliant container images:

- **Standard Image**: `casethirdpartyreader` (prod, staging)
- **FIPS Image**: `casethirdpartyreaderfips` (gov)
- **Build Process**: Separate Bazel targets for FIPS-compliant builds
- **Cryptography**: Uses FIPS-validated cryptographic modules

8.3 Configuration Management

- **Consul Integration**: Dynamic configuration via consul-template
  - Init container: `consul-template-init`
  - Template: `/etc/templates/case-third-party-reader.ini`
  - Output: `/etc/datadog/case-third-party-reader.ini`
- **Secret Management**: Integrated with Datadog's secret management system
- **Environment Isolation**: Separate configurations per environment (staging, prod, gov)
- **Tenant Support**: Multi-tenant configuration via workplatform library

9 Technical Implementation Details

9.1 Core Components

Main Entry Point (`main.go`)

- Application bootstrap using ddapp framework
- Flavor selection based on configuration (`jira` or `servicenow`)
- Kafka consumer initialization with flavor-specific topic
- Handler instantiation with integration-specific decoder, filterer, and processor
- Health check HTTP server on port 8080
- Graceful shutdown handling

Handler (`handler.go`)
Core struct:

```
type handler struct {
    decoder              thirdparty.Decoder
    filterer             thirdparty.Filterer
    processor            thirdparty.Processor
    stopper              *util.Stopper
    tagsProvider         tagsProvider
    caseClient           casepb.CasesClient
    cacheSettingsClient  caseapi.CacheSettingsClient
    flavor               string
    in                   chan *CollabIntegrationPayload
}
```

Processing method: `handler.Run(ctx context.Context)`

1. Receive payload from channel: `payload := <-h.in`
2. Decode: `changelog, err := h.decoder.Decode(payload)`
3. PreFilter: `reason, err := h.filterer.PreFilter(ctx, changelog)`
4. Lookup case: `caseResp, err := h.caseClient.Get(ctx, caseID)`
5. Filter: `reason, err := h.filterer.Filter(ctx, changelog, caseData, settings)`
6. Process: `err := h.processor.Process(ctx, changelog, caseData)`
7. Ack: `payload.Ack()`
8. Record metrics: `statsd.Distribution(...)`

Payload Wrapper (`payload.go`)

```
type CollabIntegrationPayload struct {
    Payload   json.RawMessage
    Timestamp time.Time
    ack       func()
}

func (p *CollabIntegrationPayload) Ack() {
    if p.ack != nil {
        p.ack()
    }
}
```

Separates Kafka-specific concerns from business logic.

9.2 Integration Abstractions

Decoder Interface (`libs/thirdparty/decoder.go`)

```
type Decoder interface {
    Decode(payload *Payload) (*Changelog, error)
}
```

Implementations:

- `libs/jira/decoder.go` - Decodes Jira webhook JSON
- `libs/servicenow/decoder.go` - Decodes ServiceNow diff events

Filterer Interface (`libs/thirdparty/filterer.go`)

```
type Filterer interface {
    PreFilter(ctx context.Context,
              changelog *Changelog) (FilteredReason, error)

    Filter(ctx context.Context,
           changelog *Changelog,
           caseData *Case,
           settings *Settings) (FilteredReason, error)
}
```

Two-stage design optimizes performance by deferring expensive operations.

Processor Interface (`libs/thirdparty/processor.go`)

```
type Processor interface {
    Process(ctx context.Context,
            changelog *Changelog,
            caseData *Case) error
}
```

Implementations apply integration-specific business logic.

9.3 Concurrency Model

- **Single-threaded per partition**: Each Kafka partition is processed sequentially
- **Channel-based**: Kafka consumer publishes to `in chan *CollabIntegrationPayload`
- **No concurrency within handler**: Prevents race conditions on shared state
- **Graceful shutdown**: `util.Stopper` pattern waits for in-flight messages

Tradeoff: Simplicity and correctness over parallelism (sufficient for current throughput).

9.4 Caching Strategy

Metadata Cache (Jira)

```
type metadataCache struct {
    ttl     time.Duration  // 5 minutes
    mu      sync.RWMutex
    data    map[uint64]map[metadataKey]bool
}

type metadataKey struct {
    accountID   string
    projectID   string
    issueTypeID string
}
```

Benefits:

- Avoids settings lookup for 99% of events during PreFilter
- Cache hit rate $> 95\%$ in production
- Reduces load on Settings API

Settings Cache

`CacheSettingsClient` provides in-memory caching of project settings with TTL and LRU eviction.

9.5 Build and Deployment Configuration

Bazel Build (`BUILD.bazel`)

- Go binary with PGO (Profile-Guided Optimization) in release builds
- Multi-architecture Docker images: linux-amd64, linux-arm64
- Separate FIPS-compliant image targets for gov environment
- Integration test suite with comprehensive fixtures

Helm Chart (`config/k8s/`)

- Chart version: 0.0.2
- Values files:
  - `values.yaml` - Base configuration
  - `values/flavors/jira.yaml` - Jira-specific overrides
  - `values/flavors/servicenow.yaml` - ServiceNow-specific overrides
  - `values/tenants/{tenant}/` - Tenant-specific overrides
- Templates: Deployment, ConfigMap, NetworkPolicies, ServiceAccount

10 Operational Considerations

10.1 Scalability

Current Configuration

- 1 replica per flavor deployment per datacenter
- Sequential processing per Kafka partition
- Consumer group: `case-management` (shared across case management services)

Scaling Strategy
Horizontal scaling is constrained by Kafka partition count:

- Each replica handles a subset of partitions via `cooperative-sticky` assignment
- To scale beyond current capacity:
  1. Increase Kafka topic partition count
  2. Increase `replicas` in Helm values
- Current throughput is well within capacity for 1 replica

10.2 Fault Tolerance

Message Delivery Guarantees

- **At-least-once delivery**: Messages may be reprocessed after pod failure
- **Idempotency**: Optimistic locking (case version) prevents duplicate updates
- **Offset management**: Manual commit with 5s auto-commit interval

Pod Failure Recovery

1. Pod crashes (from retry exhaustion or OOM)
2. Kubernetes restarts pod (restart policy: Always)
3. Kafka consumer rejoins consumer group
4. Uncommitted offsets are reprocessed

5. Idempotency ensures no duplicate case updates

Partition Rebalancing
`cooperative-sticky` partition assignment strategy:

- Minimizes partition movement during rebalancing
- Avoids stop-the-world rebalancing (can continue processing other partitions)
- Improves availability during scaling or pod restarts

10.3 Monitoring and Alerting

Critical Alerts
Recommended alert configurations:

1. **High Filtering Rate**
   - Metric: `dd.case_third_party_reader.jira.filtered`
   - Threshold: $> 80\%$ of events filtered for > 15 minutes
   - Indicates: Configuration issue or synchronization loop
2. **Consumer Lag**
   - Metric: Kafka consumer lag
   - Threshold: Lag $> 10{,}000$ messages for > 30 minutes
   - Indicates: Throughput issue or processing bottleneck
3. **High Error Rate**
   - Metric: `dd.case_third_party_reader.decoding_failed` or retry metric
   - Threshold: $> 5\%$ error rate for > 10 minutes
   - Indicates: Upstream payload changes or API failures
4. **Pod Crash Loop**
   - Metric: Kubernetes pod restart count
   - Threshold: > 5 restarts in 15 minutes
   - Indicates: Persistent error requiring investigation

Service Catalog

- **Team**: case-management
- **On-Call**: `https://app.datadoghq.com/on-call/teams/bc33b578-f8a9-11ed-a001-da7ad0900002`
- **Slack Channels**:
  - #case-management (primary)
  - #case-management-releases-stg (staging releases)
- **Documentation**: `https://datadoghq.atlassian.net/wiki/spaces/CM/pages/3294330939`
- **Repository**: `https://github.com/DataDog/dd-source/tree/main/domains/case_management/apps/case-third-party-reader`

11 Future Considerations

11.1 Scalability Enhancements

- **Partition-Level Autoscaling**: Dynamically adjust replica count based on per-partition lag
- **Batch Processing**: Process multiple events in a single API call (requires Case API support)
- **Async Processing**: Decouple case updates from timeline cell creation for higher throughput

11.2 Feature Extensions

- **Additional Integrations**: Support for more third-party systems (Linear, GitHub Issues, etc.)
- **Custom Field Flexibility**: Generalized custom field mapping beyond due dates
- **Conflict Resolution Strategies**: Configurable resolution strategies (last-write-wins, manual review, etc.)
- **Event Replay**: Admin tool to replay filtered events after configuration changes

11.3 Operational Improvements

- **Enhanced Observability**:
  - Per-organization latency SLOs
  - Filter reason aggregation in dashboards
  - State conflict visualization
- **Automated Testing**:
  - Integration tests against live third-party sandboxes
  - Chaos engineering for fault injection
  - Canary deployments with synthetic traffic
- **Self-Service Configuration**:
  - UI for managing field mappings and sync settings
  - Dry-run mode to preview sync behavior
  - Configuration validation before apply

11.4 Performance Optimizations

- **Cache Warming**: Pre-populate metadata caches on startup to reduce cold-start latency
- **Settings Cache TTL Tuning**: Dynamic TTL based on configuration change frequency
- **Bulk User Resolution**: Batch user lookups to reduce round-trips to User Service
- **Compression**: Enable Kafka message compression for reduced network overhead

12 Conclusion

The Case-Third-Party-Reader serves as a critical bidirectional synchronization bridge between Datadog's Case Management system and third-party ticketing platforms. Its sophisticated two-stage filtering architecture, comprehensive error handling, and integration-specific abstractions enable reliable and performant event processing at scale.

Key Architectural Strengths:

- **Flavor-Based Deployment**: Isolation between integrations prevents cascading failures
- **Two-Stage Filtering**: PreFilter optimization reduces expensive lookups by $> 80\%$
- **State Conflict Detection**: Prevents data loss from concurrent updates
- **Metadata Caching**: 5-minute TTL caches provide $> 95\%$ hit rate
- **Graceful Error Handling**: Three-tier classification (skip/retry/discard) ensures robustness
- **Idempotent Processing**: Optimistic locking enables at-least-once delivery semantics

Operational Excellence:

- **Comprehensive Observability**: 27 distinct filter reasons enable precise debugging
- **Zero-Downtime Deployments**: Rolling updates with health gates
- **Multi-Environment Support**: Staging, production, and FIPS-compliant government deployments
- **Security and Compliance**: Network isolation, RBAC, and data access controls

Integration Capabilities:

- **Jira**: Full bidirectional sync with custom field support and state conflict detection
- **ServiceNow**: Comment and status synchronization with instance-based filtering
- **Extensibility**: Abstraction interfaces (Decoder, Filterer, Processor) enable easy addition of new integrations

The service's architecture balances simplicity with robustness, preferring correctness over premature optimization. The single-threaded processing model per partition eliminates concurrency bugs while providing sufficient throughput for current scale. As Case Management adoption grows, the flavor-based deployment model and Kafka's horizontal scaling capabilities provide clear scaling paths.

This report provides a comprehensive technical overview of the Case-Third-Party-Reader service, its integration patterns, deployment architecture, and operational characteristics across Datadog's global infrastructure.

This report documents the Case-Third-Party-Reader service as of January 2026. For the latest implementation details, refer to the source code at:

`domains/case_management/apps/case-third-party-reader`