The Complete Guide to

Case Management Search

From User Query to Frontend Response

A Deep Dive into Query Processing, Elasticsearch Integration, and Result Transformation in Datadog Case Management

Generated on: September 19, 2025

Technical Documentation Series

Understanding the complete journey from user input to structured frontend data

Table of Contents

Chapter 1: Introduction to the Search Journey

When a user types a query into the Case Management interface, they initiate a complex journey through multiple layers of sophisticated software architecture. This journey transforms human-readable search terms into optimized Elasticsearch queries, executes them against massive datasets, processes the results through enrichment pipelines, and finally delivers structured data back to the frontend for presentation. Understanding this complete flow is crucial for developers, architects, and engineers working with the Case Management system. Each step in this process involves careful orchestration of parsing, validation, transformation, execution, and response formatting. The system handles three distinct types of search operations, each with its own unique characteristics and processing patterns. The first type is Basic Search, which handles straightforward queries like finding cases with specific statuses or priorities. These queries follow a direct path from user input through ANTLR parsing, Elasticsearch query building, and result formatting. The system applies security filters, handles pagination, and enriches results with human-readable names before sending responses to the frontend. The second type is Analytics Search, which powers the time-series visualizations and aggregations that users see in dashboards and reports. These queries are far more complex, involving multi-level aggregations, time bucketing, custom attribute handling, and metric calculations. The system must carefully orchestrate nested aggregations, handle custom attribute queries through Elasticsearch's nested object capabilities, and compress large result sets for efficient transmission. The third type is Facet Search, which enables discovery and autocomplete functionality. This includes retrieving available facets for filtering, searching for specific facet values with partial matching, and providing autocomplete suggestions as users type. These operations require specialized handling of Elasticsearch aggregations and careful result processing to provide fast, relevant suggestions. Throughout this guide, we will follow complete examples of each search type, examining the actual code paths, Elasticsearch queries, and response transformations. We will see how a simple user input like "status:open" becomes a complex Elasticsearch bool query with security filters, how analytics queries generate nested aggregations with time buckets, and how the system transforms raw Elasticsearch responses into the structured data that powers the frontend interfaces. Each chapter builds upon the previous one, creating a comprehensive understanding of how search works in the Case Management domain. By the end of this guide, you will understand not just what happens during a search, but why each transformation is necessary and how all the pieces fit together to create a seamless user experience.

Chapter 2: Basic Search - From Query to Results

Basic search represents the most common interaction users have with the Case Management system. When a user enters a query like "status:open AND priority:high" into the search interface, they expect to see a list of cases that match their criteria, paginated and sorted appropriately. Behind this seemingly simple operation lies a sophisticated pipeline that ensures accuracy, security, and performance. The journey begins when the user submits their query through the web interface. The frontend application constructs a SearchQuery protobuf message containing the user's input along with additional parameters like pagination settings, sort preferences, and time range filters. This message is then sent to the Case Rapid API, which validates the request and routes it to the appropriate handler. The SearchHandler receives this request and immediately begins the transformation process. The first step involves parsing the user's query string using an ANTLR-based parser that understands the custom query language developed for Case Management. This parser can handle complex boolean logic, field-specific searches, range queries, and even nested custom attribute searches. Let us follow a specific example to understand this process in detail. Consider a user who searches for "status:open AND priority:high AND assignee:john.doe@datadog.com". This query contains multiple field-specific searches combined with boolean AND logic, representing a typical user query pattern. The ANTLR parser processes this input and generates an abstract syntax tree that represents the logical structure of the query. The parser recognizes that this is a boolean AND operation with three field-value pairs: status equals open, priority equals high, and assignee equals a specific email address. Next, the ESQueryBuilder takes this parsed structure and transforms it into Elasticsearch Query DSL. For our example query, this involves creating a bool query with three must clauses. The status field becomes a term query, the priority field requires special handling because priorities are stored as numeric enum values, and the assignee field needs to support both UUID and email lookups through the user enrichment system. However, before the query is executed, the system applies crucial security filters. The queryBuilder automatically adds organization ID filtering to ensure users only see cases from their own organization. It also integrates with the ProjectService to determine which projects the user has access to, adding additional project ID filters to the query. This security layer operates transparently, ensuring that users never see data they should not have access to.

Query Transformation in Detail

To understand the transformation process, let us examine the exact steps that occur when processing our example query "status:open AND priority:high AND assignee:john.doe@datadog.com". The ANTLR parser first tokenizes this input, identifying keywords, operators, field names, and values. It recognizes "status", "priority", and "assignee" as field identifiers, "open", "high", and the email address as values, and "AND" as boolean operators. The parser then constructs an abstract syntax tree where the root node is an AND operation with three child nodes representing the field-value pairs. The ESQueryBuilder receives this syntax tree and begins the transformation to Elasticsearch Query DSL. For the status field, it creates a simple term query since status values are stored as keyword fields in Elasticsearch. The priority field requires more complex handling because priorities are stored as numeric enum values rather than strings. The system must map "high" to its corresponding numeric value before creating the term query. The assignee field presents the most complex transformation challenge. Users can search for assignees using either email addresses or UUIDs, but Elasticsearch stores only UUIDs in the assignee_id field. When the system encounters an email address in the assignee field, it must first resolve this email to a UUID using the UserService, then create a term query with the resolved UUID. The security filtering layer then adds additional query clauses. The system automatically injects a term query for the user's organization ID, ensuring they only see cases from their own organization. It also calls the ProjectService to retrieve the list of project IDs the user has access to, adding a terms query that restricts results to only those projects. The final Elasticsearch query structure becomes a complex bool query with multiple must clauses. The user's original three search terms become three must clauses, while the security filters add two additional must clauses for organization and project filtering. This ensures that the search results are both relevant to the user's query and properly scoped to their access permissions. Here is the actual Elasticsearch query that would be generated: { "query": { "bool": { "must": [{ "term": { "status": 1 } }, { "term": { "priority": 3 } }, { "term": { "assignee_id": "550e8400-e29b-41d4-a716-446655440000" } }, { "term": { "org_id": 12345 } }, { "terms": { "project_id": ["proj-1", "proj-2", "proj-3"] } }] } }, "size": 10, "from": 0, "sort": [{ "created_at": { "order": "desc" } }] } This query demonstrates how a simple user input becomes a sophisticated Elasticsearch query with proper security filtering, data type handling, and performance optimizations.

Execution and Response Processing

Once the Elasticsearch query is constructed and validated, the system executes it against the cases index. The SearchHandler configures the query with additional parameters including pagination settings, sorting preferences, and field selection options. If the user has specified that they only want certain fields returned, the system adds a source filter to reduce the amount of data transferred from Elasticsearch. The query execution occurs within a configured timeout context, typically five seconds for basic searches. This timeout ensures that slow queries do not impact system performance or user experience. The system sends the query to the Elasticsearch cluster and waits for the response. When Elasticsearch returns the search results, the response contains a hits object with detailed information about matching documents. Each hit includes the document's source data, score, and metadata. The total hits count indicates how many documents match the query across the entire index, while the returned hits represent only the current page of results. The system then begins the complex process of transforming these raw Elasticsearch documents into the structured protobuf messages that the frontend expects. This transformation involves several critical steps that ensure data consistency and user-friendly presentation. First, each Elasticsearch document is deserialized from JSON into an internal Case struct. This struct represents the Elasticsearch document structure and includes all the fields that were stored in the index. The system must carefully handle type conversions, null values, and nested objects during this deserialization process. Next, the system converts the internal Case struct into a protobuf Case message. This conversion involves mapping Elasticsearch field names to protobuf field names, handling enum conversions, and ensuring that all required fields are populated. The protobuf structure represents the canonical format for case data that all client applications expect to receive. The enrichment process then begins, which is one of the most important aspects of result processing. Many fields in the Elasticsearch documents contain UUIDs or internal identifiers that are not meaningful to users. The enrichment system resolves these identifiers into human-readable names and additional metadata that enhance the user experience. For example, the assignee_id field contains a UUID that identifies the assigned user. The enrichment system calls the UserService to resolve this UUID into the user's name and email address. Similarly, project_id values are resolved into project names, and case type identifiers are converted into descriptive case type names. The enrichment process operates efficiently by batching requests for related identifiers. Rather than making individual service calls for each UUID, the system collects all unique identifiers that need resolution and makes batch requests to the appropriate services. This approach minimizes network overhead and improves response times. Finally, the system constructs the SearchResponse protobuf message that will be returned to the client. This response includes the enriched case objects, pagination metadata such as total count and page count, and any additional context information that the frontend might need for rendering the results.

Chapter 3: Analytics Search - Time Series and Aggregations

Analytics search represents the most sophisticated and computationally intensive operation in the Case Management search system. When users interact with dashboards, generate reports, or explore trends over time, they are utilizing the powerful analytics capabilities that transform raw case data into meaningful insights through complex aggregations and time-series analysis. The analytics journey begins when a user configures a dashboard widget or runs a report that requires aggregated data. For example, a user might want to see the number of cases grouped by status over the past month, with data points every day. This seemingly simple request requires the system to perform multi-level aggregations across potentially millions of documents, handle time bucketing, and process custom attributes through Elasticsearch's nested object capabilities. Consider a specific example where a user requests to see case counts grouped by status and assignee over the past seven days, with hourly data points. The user interface constructs an AnalyticQuery message containing the search criteria, grouping parameters, time range, and aggregation settings. This query includes a base filter for recent cases, grouping by both status and assignee fields, and a request for hourly time buckets with count aggregation. The AnalyticHandler receives this request and immediately recognizes the complexity involved. Unlike basic search queries that return individual documents, analytics queries must construct sophisticated aggregation pipelines that can group, bucket, and calculate metrics across large datasets. The system must handle multi-field grouping, time bucketing, custom attribute support, and efficient result compression. The first step in analytics processing involves parsing any query filters using the same ANTLR parser used for basic searches. However, the parsed query is then integrated into a much more complex aggregation structure. The system must determine the appropriate aggregation hierarchy based on the grouping fields and time bucketing requirements. For our example query, the system constructs a multi-level aggregation pipeline. At the top level, it creates a multi-terms aggregation that groups documents by both status and assignee fields. Within each group, it creates a date range aggregation that divides the time period into hourly buckets. Finally, within each time bucket, it applies the count aggregation to determine how many cases fall into each category for each time period. The complexity increases significantly when custom attributes are involved in the grouping. Custom attributes in the Case Management system are stored as nested objects in Elasticsearch, which requires special aggregation handling. When a user wants to group by a custom attribute like "environment" or "service", the system must construct nested aggregations that first enter the custom_attributes array, filter for the specific attribute key, group by the attribute values, and then use reverse nested aggregations to access the parent document for time bucketing. The time bucketing logic itself involves sophisticated calculations to determine the appropriate bucket size and count. The system takes the requested time range and interval, calculates how many buckets would be created, and ensures that the total does not exceed configured limits. If the requested granularity would create too many buckets, the system automatically adjusts to a larger interval while maintaining the overall time coverage.

Complex Aggregation Architecture

To understand the sophistication of analytics aggregations, let us examine the exact Elasticsearch query structure that would be generated for our example: case counts grouped by status and assignee over seven days with hourly buckets. The system begins by constructing the base query with time range and security filters, similar to basic search. However, instead of returning document hits, the query sets the size parameter to zero and focuses entirely on aggregations. The aggregation structure becomes the heart of the query, containing multiple nested levels that work together to produce the desired grouping and time series data. At the root level, the system creates a multi_terms aggregation that groups documents by both status and assignee_id fields simultaneously. This aggregation tells Elasticsearch to create buckets where each bucket represents a unique combination of status and assignee values. The system configures this aggregation with appropriate size limits and sorting to ensure performance and relevance. Within each multi_terms bucket, the system adds a date_range aggregation that divides the seven-day time period into hourly intervals. The date range calculation involves creating 168 individual time ranges (24 hours × 7 days), each representing a one-hour period. Each range specifies exact from and to timestamps in milliseconds, ensuring precise time bucketing. The resulting Elasticsearch aggregation structure looks like this: {   "aggs": { "root_aggr": {   "multi_terms": {   "terms": [{ "field": "status" },   { "field": "assignee_id" }   ],   "size": 100 },   "aggs": {   "time_bucket_aggr": {   "date_range": { "field": "created_at",   "ranges": [{ "from": 1640995200000, "to": 1640998800000 },   { "from": 1640998800000, "to": 1641002400000 },   // ... 166 more hourly ranges   ] }   }   }   }   } } This structure instructs Elasticsearch to first group all matching documents by status-assignee combinations, then within each group, distribute the documents across hourly time buckets. The result provides exactly the data needed to render a time-series chart showing case creation trends for different status-assignee combinations. When custom attributes are involved in the grouping, the aggregation structure becomes even more complex. Consider a query that groups by a custom attribute called "environment". The system must construct a nested aggregation that handles the custom_attributes array structure. The nested aggregation first enters the custom_attributes array context, then applies a filter aggregation to find documents where the custom attribute key equals "environment". Within this filtered context, it creates a terms aggregation on the custom attribute value, then uses a reverse_nested aggregation to return to the parent document context for time bucketing. This nested structure ensures that the system can group by custom attribute values while still maintaining access to the parent document's timestamp and other fields needed for time bucketing and additional filtering.

Result Processing and Compression

When Elasticsearch returns the aggregation results, the response contains a complex nested structure that mirrors the aggregation hierarchy. The AnalyticHandler must carefully traverse this structure, extracting the relevant data points and transforming them into the time-series format expected by the frontend visualization components. The extraction process begins at the root aggregation and works its way down through each level. For our example query with status-assignee grouping, the system first extracts the multi_terms buckets, each representing a unique status-assignee combination. The bucket key contains an array with the status value and assignee UUID, while the bucket contains the nested time_bucket_aggr aggregation. For each group bucket, the system then extracts the date_range aggregation buckets, each representing an hourly time period. These buckets contain the document counts that fell within each time range for the specific status-assignee combination. The system builds a time-series array where each element corresponds to an hourly time slot, with values representing the case counts for that particular group. The challenge lies in handling missing data points and ensuring consistent time series structure across all groups. Not every status-assignee combination will have cases in every hourly bucket, so the system must fill in zero values for missing time periods. This ensures that all groups have time series data with the same length and corresponding time points, which is essential for proper visualization rendering. The enrichment process for analytics results operates similarly to basic search but at a different scale. Instead of enriching individual case documents, the system must enrich the group keys used in the aggregations. For our example, this means resolving assignee UUIDs to user names and email addresses, and converting status enum values to human-readable status names. The enrichment system batches these requests efficiently, collecting all unique UUIDs and enum values across all aggregation buckets before making service calls. This approach minimizes network overhead even when dealing with hundreds of unique group combinations. Once the data is extracted and enriched, the system faces the challenge of efficiently transmitting potentially large result sets to the frontend. Analytics queries can generate thousands of data points across hundreds of groups, creating response payloads that could impact network performance and user experience. The system addresses this through sophisticated compression and optimization techniques. The time series data is structured as separate arrays for epochs (time points), groups (series names), and values (data points). This structure eliminates redundant timestamp data and enables efficient compression algorithms. The MetricsBuffer format uses LZ4 compression to reduce the response payload size significantly. The system compresses the epochs array (timestamp data), values array (numeric data), and includes metadata about group counts and time bucket counts. This compressed format can reduce response sizes by 70-80% while maintaining fast decompression on the frontend. The final response structure provides everything the frontend needs to render interactive time-series visualizations. It includes the compressed data arrays, group labels with human-readable names, time range metadata, and aggregation method information. The frontend can efficiently decompress this data and feed it directly into charting libraries without additional processing.

Chapter 4: Facet Search - Discovery and Autocomplete

Facet search enables users to discover and explore the available search dimensions within their case data. When users interact with search filters, dropdown menus, or autocomplete suggestions, they are utilizing sophisticated facet search capabilities that help them understand what data is available and construct more effective queries. The facet search system operates through several specialized handlers that work together to provide discovery and suggestion functionality. The GetFacetsHandler returns the list of available facets that users can search by, while the SearchFacetValuesHandler enables users to find specific values within those facets. The SearchAutocompleteHandler provides real-time suggestions as users type their queries. Consider the user experience when someone wants to filter cases by assignee. They might start by requesting the list of available facets, which would include "assignee" among other options like "status", "priority", "project", and "service". When they select the assignee facet, the system needs to provide a list of actual assignee values that exist in their accessible cases, potentially with search and filtering capabilities if there are many assignees. The GetFacetsHandler begins this process by returning a predefined list of supported facet fields. This list includes both standard case fields like status, priority, and assignee, as well as standard attributes like service, team, and version. The handler also supports custom attributes, which requires special handling to discover the available custom attribute keys dynamically. For standard facets, the system returns a static list of supported fields along with metadata about their types and characteristics. For custom attributes, the system must query Elasticsearch to discover what custom attribute keys actually exist in the user's accessible cases. This involves a nested aggregation that extracts unique custom attribute keys from the custom_attributes array. When users want to search for specific values within a facet, the SearchFacetValuesHandler takes over. This handler constructs specialized Elasticsearch queries that can efficiently find and return facet values, often with partial matching and autocomplete functionality. For a query like "find assignees whose names contain 'john'", the system constructs an aggregation-based query that can search through assignee values and return matching results. However, this process is complicated by the fact that assignee_id fields contain UUIDs rather than human-readable names. The system must therefore integrate with the UserService to resolve UUIDs to names and perform the actual search against the resolved names. The SearchAutocompleteHandler provides real-time suggestions as users type. When a user types "stat" into a search field, the system recognizes this as a potential field name and returns suggestions like "status". When they continue typing "status:op", the system recognizes they are looking for status values and suggests "status:open", "status:opened", etc. This autocomplete functionality requires sophisticated parsing and prediction logic that can understand partial queries, suggest completions for both field names and values, and handle complex query structures with boolean operators and nested expressions.

Chapter 5: Advanced Query Patterns

The Case Management search system supports sophisticated query patterns that enable users to construct complex searches involving custom attributes, boolean logic, date ranges, and nested object queries. Understanding these patterns is crucial for developers who need to extend the system or troubleshoot complex search scenarios. Custom attribute queries represent one of the most complex patterns in the system. When a user searches for "custom_attributes.environment:production", they are requesting cases where a specific custom attribute has a particular value. This seemingly simple query requires the system to construct a nested Elasticsearch query that can search within the custom_attributes array. The transformation process begins with the ANTLR parser recognizing the custom_attributes prefix and treating the remainder as a nested attribute query. The ESQueryBuilder then constructs a nested query that first enters the custom_attributes context, applies a term filter for the attribute key ("environment"), and then searches for the specified value ("production") in either the value_text or value_number fields. The complexity increases when users combine custom attribute queries with other search terms. A query like "status:open AND custom_attributes.environment:production AND priority:high" requires the system to construct a boolean query that combines standard field searches with nested custom attribute searches. Date range queries provide another layer of sophistication. Users can specify absolute dates like "created_at:[2023-01-01 TO 2023-12-31]" or relative dates like "created_at:[now-7d TO now]". The system must parse these date expressions, handle timezone conversions, and construct appropriate Elasticsearch range queries. The relative date parsing is particularly complex because it must support various time unit expressions (s, m, h, d, w, M, y) and handle calculations relative to the current time. The system also needs to validate that date ranges make sense and apply reasonable limits to prevent queries that span excessive time periods. Boolean logic handling requires careful precedence management and parentheses support. A query like "(status:open OR status:in_progress) AND (priority:high OR priority:critical) AND assignee:john.doe@example.com" demonstrates the complexity of nested boolean expressions that the parser must handle correctly. The ANTLR grammar defines operator precedence rules that ensure AND operations bind more tightly than OR operations, while parentheses can override default precedence. The resulting syntax tree must properly represent the user's intended logic structure, which the ESQueryBuilder then transforms into appropriately nested bool queries in Elasticsearch. Range queries extend beyond dates to include numeric fields and support various comparison operators. Users can search for cases with specific comment counts using queries like "comment_count:>5" or "comment_count:[1 TO 10]". The system must recognize these patterns and construct the appropriate range queries with the correct boundary conditions.

Chapter 6: Elasticsearch Deep Dive

The Elasticsearch integration in the Case Management system represents a sophisticated implementation that leverages advanced features of the Elasticsearch platform. Understanding the specific patterns, optimizations, and design decisions helps explain why the system performs well under load and handles complex queries efficiently. The cases index uses a carefully designed mapping that balances search performance with storage efficiency. The mapping disables dynamic field creation to maintain strict schema control, ensuring that all fields have appropriate types and analyzers. This approach prevents mapping explosions that can occur when dynamic content creates numerous unexpected fields. Text fields like title and description use standard analyzers that support full-text search with stemming, lowercasing, and stop word removal. These fields also maintain keyword sub-fields that enable exact matching and aggregations. This dual-field approach provides both search flexibility and aggregation performance. Keyword fields like internal_id, case_id, and assignee_id use exact matching without analysis. These fields support fast term queries and efficient aggregations, making them ideal for filtering and grouping operations. The keyword type also supports doc values, which enable memory-efficient sorting and field data access. Date fields use Elasticsearch's optimized date handling with millisecond precision. The system stores all timestamps in UTC and relies on Elasticsearch's date math capabilities for range queries and time zone handling. Date fields support efficient range queries and date histogram aggregations used in analytics. The custom_attributes field uses the nested object type, which is crucial for proper query behavior. Unlike regular object fields that are flattened, nested fields maintain the relationship between custom attribute keys and values. This enables precise queries that can find cases where a specific custom attribute key has a specific value, rather than matching documents where any key-value combination exists. Long fields store numeric identifiers and enum values efficiently. These fields support range queries, aggregations, and fast filtering operations. The system uses long fields for org_id, status, priority, and type_id, enabling efficient filtering and grouping by these common facets. The flattened field type handles the attributes object, which contains standard case attributes that don't require nested querying. Flattened fields provide a balance between flexibility and performance for semi-structured data that needs to be searchable but doesn't require complex relationships. Query construction follows patterns optimized for Elasticsearch performance. Bool queries with must clauses enable Elasticsearch to use efficient intersection algorithms. Term queries on keyword fields benefit from fast exact matching. Range queries on date and numeric fields use optimized data structures for quick range identification. Aggregations leverage Elasticsearch's sophisticated aggregation framework. Terms aggregations benefit from field data caching and efficient bucket generation. Date range aggregations use optimized time-based indexing structures. Nested aggregations carefully manage context switching to maintain performance while providing accurate results for custom attributes. The system implements query optimization strategies including field selection to reduce network transfer, appropriate use of source filtering to limit returned data, and careful timeout management to prevent slow queries from impacting system performance.

Chapter 7: Result Processing and Enrichment

The transformation of raw Elasticsearch responses into user-friendly frontend data represents one of the most critical aspects of the search system. This process involves multiple stages of parsing, validation, enrichment, and formatting that ensure users receive accurate, complete, and meaningful information. When Elasticsearch returns search results, the response contains raw JSON documents that represent the stored case data. These documents use internal identifiers, enum values, and technical field names that are not suitable for direct presentation to users. The system must transform this technical data into human-readable information while maintaining data integrity and relationships. The deserialization process begins by parsing the Elasticsearch JSON response into strongly-typed Go structs. The elasticsearch.Case struct defines the exact structure of documents stored in the index, including all field types, nested objects, and array structures. This deserialization process includes validation to ensure that the received data matches expected formats and contains required fields. During deserialization, the system handles type conversions carefully. Numeric fields stored as JSON numbers must be converted to appropriate Go types. Date fields stored as ISO timestamp strings or millisecond integers must be parsed into time.Time values. Array fields must be properly handled to avoid nil pointer exceptions when documents contain missing or empty arrays. The protobuf conversion stage transforms the internal structs into the canonical protobuf message format used throughout the Case Management system. This conversion involves mapping field names from Elasticsearch conventions to protobuf conventions, handling enum conversions where numeric values must be mapped to protobuf enum types, and ensuring that optional fields are properly set or cleared. Enum handling requires special attention because Elasticsearch stores enum values as integers while protobuf represents them as named constants. The system maintains mapping tables that convert between these representations, handling cases where new enum values might exist in the data but not yet in the protobuf definitions. The enrichment process represents the most user-visible transformation stage. Many fields in the case documents contain UUIDs or internal identifiers that have no meaning to users. The assignee_id field contains a UUID that identifies a user, but users need to see names and email addresses. Project_id fields contain project identifiers that need to be resolved to project names. The enrichment system operates through a series of specialized services. The UserService integrates with the OUI (Organization User Interface) system to resolve user UUIDs into user profiles containing names, email addresses, and other relevant information. The ProjectService resolves project identifiers into project metadata including names, descriptions, and access permissions. Batch processing optimization ensures that enrichment operations perform efficiently even when processing large result sets. Rather than making individual service calls for each UUID, the system collects all unique identifiers that need resolution and makes batch requests. This approach reduces network overhead and service load while improving response times. The enrichment system handles failures gracefully by providing fallback behavior when service calls fail or return incomplete data. If a user UUID cannot be resolved, the system retains the UUID in the response rather than failing the entire request. This approach ensures that users receive as much information as possible even when some enrichment operations fail. Response formatting creates the final structure that frontends receive. The system constructs protobuf response messages that include enriched case objects, pagination metadata, search statistics, and any error information that clients might need for proper handling.

Chapter 8: Error Handling and Edge Cases

Robust error handling throughout the search pipeline ensures that users receive meaningful feedback when queries fail and that the system degrades gracefully under various failure conditions. The search system encounters numerous potential failure points, from invalid user input to Elasticsearch cluster issues, and must handle each scenario appropriately. Query validation begins at the earliest stage when user input is received. The ANTLR parser can encounter syntax errors when users provide malformed queries with unbalanced parentheses, invalid operators, or unrecognized field names. When parsing failures occur, the system returns specific error messages that help users understand and correct their query syntax. Field validation ensures that users can only search fields that actually exist and are searchable. When a user attempts to search a non-existent field like "invalid_field:value", the system recognizes this during the parsing stage and returns an appropriate error message. This validation prevents queries that would return no results due to field name typos. Value validation checks that field values are appropriate for their field types. Date fields require valid date formats, numeric fields require numeric values, and enum fields must use valid enum values. When validation fails, the system provides specific guidance about the expected format or valid values for the field. Elasticsearch connection and timeout handling addresses network and cluster-level issues. The search system configures appropriate timeouts for different query types, with longer timeouts for complex analytics queries and shorter timeouts for basic searches. When queries exceed their timeout limits, the system returns timeout errors rather than allowing indefinite waiting. Elasticsearch query errors can occur when queries are too complex, use too much memory, or encounter internal cluster issues. The system parses Elasticsearch error responses and translates technical error messages into user-friendly explanations when possible. For example, "too_many_buckets_exception" errors are translated to messages about reducing the time range or grouping granularity. Permission and access control errors occur when users attempt to access cases or projects they don't have permissions to view. The security filtering system normally prevents these scenarios by adding appropriate filters to queries. However, edge cases like permission changes during query execution require careful handling to ensure users receive appropriate error messages. Service dependency failures affect the enrichment process when external services like UserService or ProjectService are unavailable. The system implements fallback strategies that allow searches to complete with partial enrichment rather than failing entirely. Users receive their search results with UUIDs instead of resolved names, along with information about which enrichment operations failed. Large result set handling addresses scenarios where queries return more data than the system can efficiently process or transmit. Analytics queries can potentially generate enormous result sets that exceed memory limits or network transmission capabilities. The system implements limits on aggregation bucket counts, result set sizes, and response payload sizes to prevent resource exhaustion. Invalid aggregation configurations can occur when users request analytics queries with parameters that don't make sense or would be computationally expensive. The system validates analytics requests to ensure that time ranges are reasonable, grouping fields are valid, and bucket counts are within acceptable limits. Graceful degradation strategies ensure that partial failures don't prevent users from accessing any data. When some shards are unavailable, Elasticsearch can return partial results with warnings. The search system propagates these warnings to users while still providing the available data, allowing them to make informed decisions about whether the partial results are sufficient for their needs.

Appendix A: Complete Query Examples

```
BASIC SEARCH EXAMPLES:

1. Simple field search:
   Input: "status:open"
   Result: Find all cases with open status

2. Boolean AND query:
   Input: "status:open AND priority:high"
   Result: Find cases that are both open and high priority

3. Boolean OR query:
   Input: "status:open OR status:in_progress"
   Result: Find cases that are either open or in progress

4. Field with email value:
   Input: "assignee:john.doe@datadog.com"
   Result: Find cases assigned to specific user

5. Date range query:
   Input: "created_at:[now-7d TO now]"
   Result: Find cases created in the last 7 days

6. Custom attribute query:
   Input: "custom_attributes.environment:production"
   Result: Find cases with custom environment attribute set to production

ANALYTICS SEARCH EXAMPLES:

1. Count by status over time:
   Query: Group by status, last 30 days, daily buckets, count aggregation
   Result: Time series showing case counts by status per day

2. Multi-field grouping:
   Query: Group by status and assignee, last 7 days, hourly buckets
   Result: Time series showing case counts for each status-assignee combination

3. Custom attribute analytics:
   Query: Group by custom_attributes.service, last 24 hours, hourly buckets
   Result: Time series showing case counts by service custom attribute

4. Percentile metrics:
   Query: Group by project, pc95 aggregation on resolution_time metric
   Result: 95th percentile resolution times by project

FACET SEARCH EXAMPLES:

1. Get available facets:
   Request: List all searchable fields
   Response: [status, priority, assignee, project, service, team, ...]

2. Search facet values:
   Input: Search assignees containing "john"
   Response: List of assignee names and emails matching "john"

3. Autocomplete suggestions:
   Input: User types "stat"
   Response: ["status:", "status:open", "status:closed", ...]

4. Custom attribute facet discovery:
   Request: Find available custom attribute keys
   Response: [environment, service, team, region, ...]
```

Appendix B: Elasticsearch JSON Examples

```
BASIC SEARCH QUERY:
{
  "query": {
    "bool": {
      "must": [
        { "term": { "status": 1 } },
        { "term": { "priority": 3 } },
        { "term": { "org_id": 12345 } },
        { "terms": { "project_id": ["proj-1", "proj-2"] } }
      ]
    }
  },
  "size": 10,
  "from": 0,
  "sort": [{ "created_at": { "order": "desc" } }]
}

CUSTOM ATTRIBUTE QUERY:
{
  "query": {
    "bool": {
      "must": [
        {
          "nested": {
            "path": "custom_attributes",
            "query": {
              "bool": {
                "must": [
                  { "term": { "custom_attributes.key": "environment" } },
                  { "term": { "custom_attributes.value_text.keyword": "production" } }
                ]
              }
            }
          }
        },
        { "term": { "org_id": 12345 } }
      ]
    }
  }
}

ANALYTICS AGGREGATION:
{
  "size": 0,
  "query": { /* base query */ },
  "aggs": {
    "root_aggr": {
      "multi_terms": {
        "terms": [
          { "field": "status" },
          { "field": "assignee_id" }
        ]
      },
      "aggs": {
        "time_bucket_aggr": {
          "date_range": {
            "field": "created_at",
            "ranges": [
              { "from": 1640995200000, "to": 1640998800000 },
              { "from": 1640998800000, "to": 1641002400000 }
            ]
          }
        }
      }
    }
  }
}
```

Appendix C: Response Structure Examples

```
BASIC SEARCH RESPONSE:
{
  "cases": [
    {
      "internal_id": "case-123",
      "public_id": "CASE-456",
      "title": "Database Connection Issues",
      "status": "OPEN",
      "priority": "HIGH",
      "assignee": {
        "uuid": "550e8400-e29b-41d4-a716-446655440000",
        "name": "John Doe",
        "email": "john.doe@datadog.com"
      },
      "project": {
        "id": "proj-1",
        "name": "Production Infrastructure"
      },
      "created_at": "2023-12-01T10:30:00Z"
    }
  ],
  "total_case_count": 150,
  "total_page_count": 15
}

ANALYTICS RESPONSE:
{
  "num_groups": 4,
  "num_epochs": 168,
  "groups": [
    "status:open□assignee:john.doe@datadog.com",
    "status:open□assignee:jane.smith@datadog.com",
    "status:in_progress□assignee:john.doe@datadog.com",
    "status:closed□assignee:jane.smith@datadog.com"
  ],
  "epochs_buf": "<compressed_timestamp_data>",
  "values_buf": "<compressed_metric_data>",
  "has_data_buf": "<compressed_presence_data>"
}

FACET VALUES RESPONSE:
{
  "facet_values": [
    {
      "value": "john.doe@datadog.com",
      "display_name": "John Doe",
      "count": 25
    },
    {
      "value": "jane.smith@datadog.com",
      "display_name": "Jane Smith",
      "count": 18
    }
  ],
  "total_count": 43
}
```