

# SearchFacetValues

## Deep Dive Guide

*Complete End-to-End Analysis*

**Understanding Parameters, Processing Flow, Elasticsearch Integration, and Result Generation in Case Management Search**

### **What You'll Learn**

- Parameter-by-parameter breakdown with examples
- Complete request processing flow with diagrams
  - Elasticsearch query generation and execution
  - Real-world use cases and practical applications
- Performance considerations and optimization tips

Generated on: September 19, 2025

*Datadog Case Management Technical Documentation*

# SearchFacetValues Overview

The SearchFacetValues endpoint is a sophisticated facet search system that enables users to discover and explore available values within specific fields (facets) of their case data. This endpoint serves as the backbone for dropdown menus, autocomplete suggestions, and filter discovery in the Case Management user interface. Unlike basic search operations that return individual case documents, SearchFacetValues performs aggregation-based queries that extract unique values from specified fields across filtered datasets. This approach provides users with contextual facet values that are relevant to their current search scope, dramatically improving the search and filtering experience. The endpoint operates on the same Elasticsearch cases index used by other search operations, but instead of returning document hits, it returns aggregated facet values with their occurrence counts. This design enables efficient discovery of available filter options while maintaining high performance even on large datasets. The system supports six key parameters that work together to provide precise control over facet value extraction: `org_id` for security scoping, `query` for base dataset filtering, `facet` for field selection, `filter` for partial value matching, `limit` for result count control, and `work_type` for content type isolation. Each parameter serves a specific purpose in the facet extraction pipeline. The `org_id` ensures proper data isolation and security. The `query` parameter acts as a base filter, allowing users to discover facet values within specific subsets of their data. The `facet` parameter specifies which field to extract values from, supporting both standard case fields and complex nested structures like custom attributes. The `filter` parameter enables partial matching on the extracted values themselves, perfect for autocomplete and search-as-you-type functionality. The `limit` parameter controls response size, while `work_type` ensures proper content scoping. The processing flow involves multiple sophisticated steps: request validation, query parsing using ANTLR grammar, security filter application, Elasticsearch aggregation construction, query execution with timeout handling, result extraction and processing, value enrichment through external services, and final response formatting. Each step includes comprehensive error handling and performance optimizations. This endpoint demonstrates the power of Elasticsearch aggregations combined with intelligent query processing, security filtering, and user experience optimization. Understanding its operation provides insight into modern search system design and the complexities involved in providing intuitive, secure, and performant facet search capabilities.

# Request Structure & Parameters

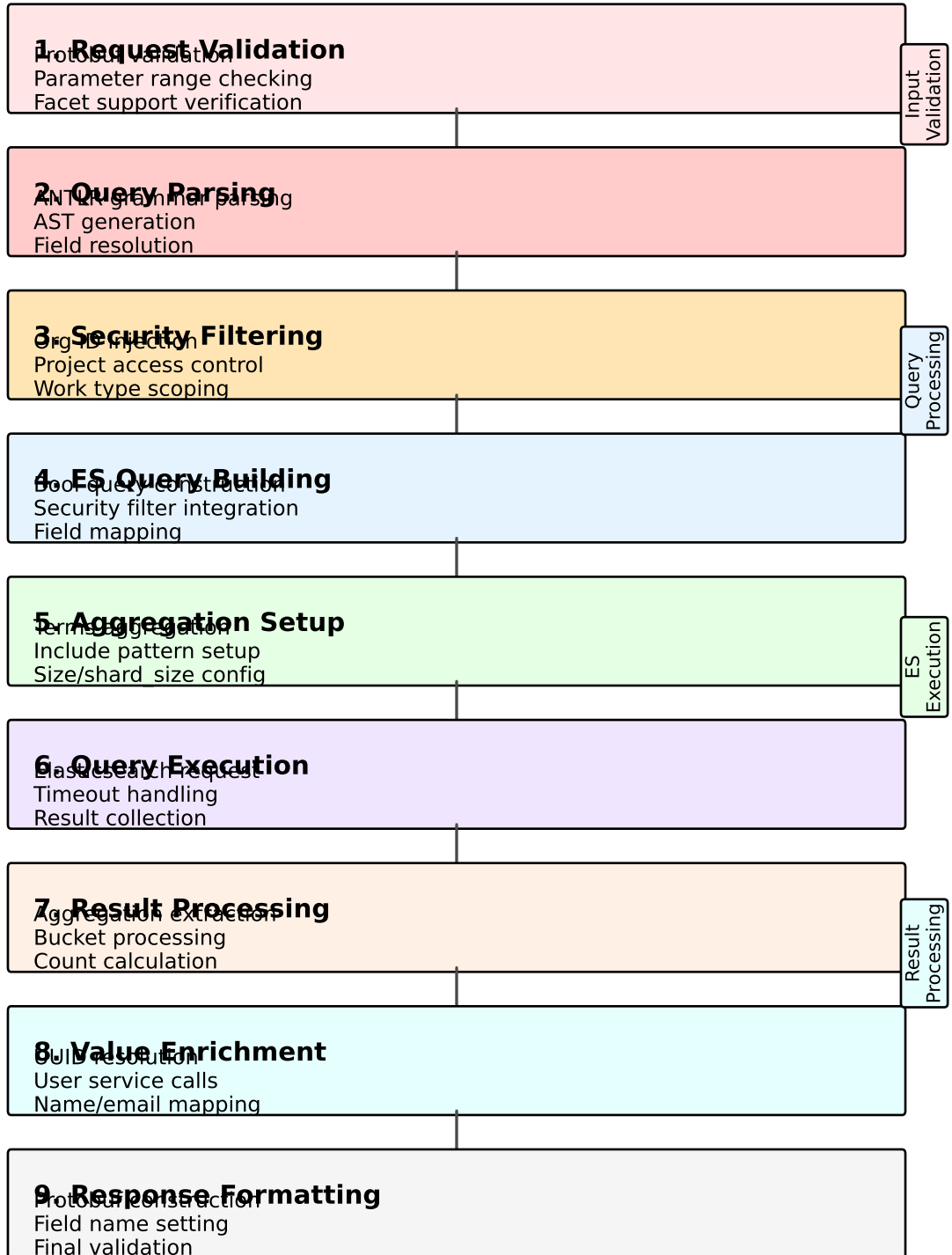
## SearchFacetValuesRequest Structure:

```
message SearchFacetValuesRequest {  
  uint64 org_id = 1;           // Organization identifier (required)  
  string query = 2;             // Base search filter (optional)  
  string facet = 3;             // Facet field to extract (required, min_len=1)  
  string filter = 4;            // Partial value matching (optional)  
  int32 limit = 5;              // Result count limit (optional, 1-100)  
  WorkType work_type = 6;       // Content type scope (optional)  
}
```

## PARAMETER BREAKDOWN:

1. org\_id (uint64, required)  
PURPOSE: Organization identifier for data isolation  
SECURITY: Automatically added as base filter to all queries  
USAGE: Used for user enrichment and permission checking  
EXAMPLE: 12345
2. query (string, optional)  
PURPOSE: Base dataset filter before facet extraction  
PARSING: Full ANTLR grammar support (same as search)  
SCOPE: Only cases matching this query contribute facet values  
EXAMPLES: "status:open", "priority:high AND created\_at:[now-7d TO now]"
3. facet (string, required, min\_len=1)  
PURPOSE: Specifies which field to extract unique values from  
VALIDATION: Must be in supported facets list  
MAPPING: Converted to actual Elasticsearch field path  
EXAMPLES: "assignee", "status", "service", "custom\_attributes.environment"
4. filter (string, optional)  
PURPOSE: Partial matching on extracted facet values  
IMPLEMENTATION: Elasticsearch terms aggregation include pattern  
OPTIMIZATION: Uses prefix patterns for short filters, wildcards for long  
EXAMPLES: "john" matches "john.doe@datadog.com", "johnny.smith@datadog.com"
5. limit (int32, optional, range: 1-100)  
PURPOSE: Controls number of facet values returned  
DEFAULT: 50 if not specified  
PERFORMANCE: Uses 2\*limit for shard\_size to improve accuracy  
EXAMPLES: 10, 25, 50
6. work\_type (WorkType, optional)  
PURPOSE: Scopes search to specific content types  
DEFAULT: CASE if not specified  
ISOLATION: Ensures API separation between work types  
EXAMPLES: CASE, INCIDENT

# SearchFacetValues Processing Flow



# Elasticsearch Integration Details

## INDEX AND DOCUMENT STRUCTURE

Target Index: "cases-read"  
Document Type: Case documents (elasticsearch.Case struct)  
Query Size: 0 (aggregation-only, no document hits returned)

The SearchFacetValues endpoint operates exclusively on case documents stored in the Elasticsearch cases index. Each facet value extraction is performed through sophisticated aggregations that analyze the fields within these case documents.

## ELASTICSEARCH QUERY STRUCTURE

The system generates complex Elasticsearch queries that combine multiple components:

Base Query Structure:

```
{
  "query": {
    "bool": {
      "must": [
        // User's parsed query clauses
        {"term": {"status": 1}},
        {"term": {"priority": 3}},
        // Security filters (auto-added)
        {"term": {"org_id": 12345}},
        {"terms": {"project_id": ["proj-1", "proj-2"]}}
      ]
    }
  },
  "size": 0, // No document hits, only aggregations
  "aggs": {
    "facet_values": {
      "terms": {
        "field": "assignee_id",
        "size": 50,
        "shard_size": 100,
        "include": ".*john.*" // From filter parameter
      }
    }
  }
}
```

## FIELD MAPPING SYSTEM

The facet parameter undergoes sophisticated field mapping to convert user-friendly facet names to actual Elasticsearch field paths:

Standard Field Mappings:

- "status" → "status" (enum stored as integer)
- "priority" → "priority" (enum stored as integer)
- "assignee" → "assignee\_id" (UUID field)
- "project" → "project\_id" (UUID field)
- "service" → "attributes.service" (flattened object field)
- "team" → "attributes.team" (flattened object field)
- "responder.name" → "additional\_properties.on\_call.responder\_users.uuid"
- "responder.email" → "additional\_properties.on\_call.responder\_users.uuid"

Custom Attribute Handling:

Custom attributes like "custom\_attributes.environment" require special nested aggregation handling due to the nested object structure in Elasticsearch. The system automatically detects these patterns and constructs appropriate nested aggregations.

## AGGREGATION OPTIMIZATION

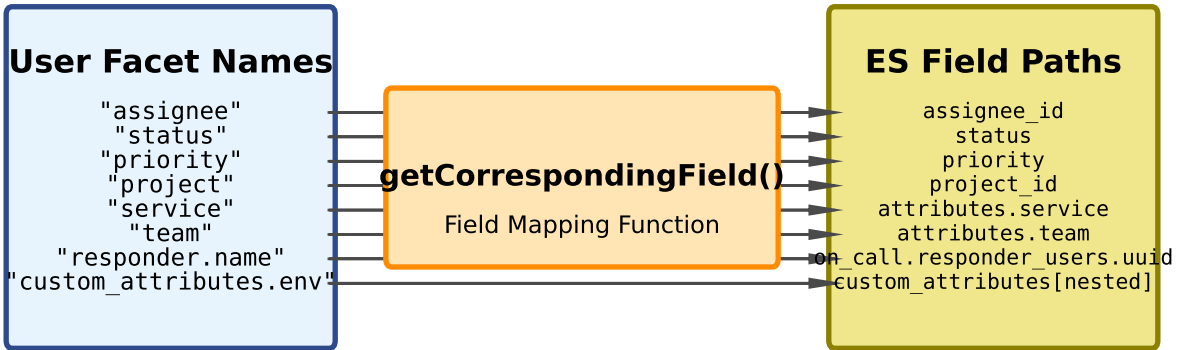
The system implements several optimization strategies:

1. Execution Hint: Uses "map" execution hint for better performance on high-cardinality fields
2. Shard Size: Sets shard\_size to 2\*limit for improved accuracy across shards
3. Include Patterns: Optimizes partial matching patterns based on filter length
4. Field Data: Leverages doc values for memory-efficient aggregations

## PERFORMANCE CONSIDERATIONS

Query Timeout: Configurable timeout (default 5 seconds) with context cancellation  
Memory Management: Aggregations use doc values to minimize heap usage  
Shard Distribution: Shard size optimization ensures accurate results across distributed shards  
Pattern Efficiency: Short filters use prefix patterns, longer filters use wildcard patterns

# Facet Field Mapping Architecture



## Special Field Handling

### RESPONDER FIELDS: UUID Resolution Required

- responder.name/email → additional\_properties.on\_call.responder\_users.uuid
- System queries UUIDs, then resolves to names/emails via UserService
- Batch processing for efficiency

### CUSTOM ATTRIBUTES: Nested Aggregation

- custom\_attributes.\* → Nested object in custom\_attributes array
- Requires nested aggregation with filter on attribute key
- Supports both text and numeric custom attribute values

### ENUM FIELDS: Value Translation

- status, priority → Stored as integers, displayed as enum names
- System converts between numeric values and human-readable names
- Maintains performance while providing user-friendly results

### FLATTENED OBJECTS: Direct Path Access

- service, team, version → attributes.{field\_name}
- Stored in flattened object for efficient querying
- Direct field access without nested complexity

# Real-World Example Scenarios

## SCENARIO 1: ASSIGNEE DISCOVERY FOR OPEN HIGH-PRIORITY CASES

Request:

```
{
  "org_id": 12345,
  "query": "status:open AND priority:high",
  "facet": "assignee",
  "filter": "john",
  "limit": 10
}
```

Processing Flow:

1. Parse "status:open AND priority:high" using ANTLR
2. Convert to ES bool query with term filters
3. Add org\_id and project security filters
4. Create terms aggregation on "assignee\_id" field
5. Apply include pattern ".\*john.\*" for partial matching
6. Execute aggregation against filtered case documents
7. Resolve assignee UUIDs to names/emails via UserService
8. Return enriched assignee list with document counts

Response:

```
{
  "field_name": "assignee",
  "values": [
    {"field_value": "john.doe@datadog.com", "count": 15},
    {"field_value": "johnny.smith@datadog.com", "count": 8},
    {"field_value": "john.wilson@datadog.com", "count": 5}
  ]
}
```

Use Case: Dropdown filter showing assignees for critical cases

## SCENARIO 2: SERVICE DISCOVERY FOR RECENT PRODUCTION CASES

Request:

```
{
  "org_id": 12345,
  "query": "created_at:[now-7d TO now] AND custom_attributes.environment:production",
  "facet": "service",
  "filter": "",
  "limit": 25
}
```

Processing Flow:

1. Parse complex query with date range and custom attribute
2. Generate nested query for custom\_attributes.environment
3. Combine with date range query on created\_at field
4. Apply security filters for org and projects
5. Create terms aggregation on "attributes.service" field
6. Execute without include filter (empty filter parameter)
7. Return all service values from matching cases

Response:

```
{
  "field_name": "service",
  "values": [
    {"field_value": "user-service", "count": 45},
    {"field_value": "payment-service", "count": 32},
    {"field_value": "notification-service", "count": 28}
  ]
}
```

Use Case: Understanding which services had issues in production recently

## SCENARIO 3: PRIORITY AUTOCOMPLETE

Request:

```
{
  "org_id": 12345,
  "query": "",
  "facet": "priority",
  "filter": "hi",
  "limit": 5
}
```

Processing Flow:

1. Empty query means consider all accessible cases
2. Apply only security filters (org\_id, project access)
3. Create terms aggregation on "priority" field
4. Apply include pattern "hi.\*" (prefix pattern for short filter)
5. Execute aggregation across all case documents
6. Convert priority enum values to human-readable names
7. Return matching priority values

Response:

```
{
  "field_name": "priority",
  "values": [
    {"field_value": "HIGH", "count": 156},
    {"field_value": "HIGHEST", "count": 23}
  ]
}
```

Use Case: Autocomplete suggestions as user types "hi" in priority filter

# Performance Optimization & Best Practices

## ELASTICSEARCH AGGREGATION OPTIMIZATIONS

**Execution Hints:** The system uses "map" execution hint for terms aggregations, which provides better performance on high-cardinality fields by using field data caching instead of ordinals.

**Shard Size Configuration:** Sets shard\_size to 2\*limit to improve accuracy across distributed shards. This ensures that each shard returns more candidates than the final limit, allowing for better global sorting and more accurate top-N results.

**Include Pattern Optimization:** Filters are optimized based on length - short filters (< 3 chars) use efficient prefix patterns while longer filters use wildcard patterns. This balances functionality with performance.

**Doc Values Usage:** All aggregated fields use doc values for memory-efficient operations, avoiding heap-based fielddata and enabling disk-based aggregations that scale better with large datasets.

## QUERY PROCESSING OPTIMIZATIONS

**ANTLR Parser Caching:** The query parser benefits from internal caching of grammar and AST structures, reducing parse time for common query patterns.

**Security Filter Integration:** Rather than post-filtering results, security constraints are integrated directly into the base query, reducing the dataset size before aggregation.

**Field Mapping Cache:** Field name to Elasticsearch path mappings are cached in memory to avoid repeated string processing and lookup operations.

## NETWORK AND SERIALIZATION OPTIMIZATIONS

**Response Size Control:** The limit parameter directly controls both network payload size and processing time, with validation ensuring reasonable bounds (1-100 values).

**Minimal Field Selection:** Aggregation queries set size=0 to avoid transferring document source data, focusing bandwidth on aggregation results only.

**Batch User Resolution:** When resolving UUIDs to user names, the system batches all unique UUIDs in a single service call rather than making individual requests.

**Connection Pooling:** The Elasticsearch client uses connection pooling and keep-alive to minimize connection overhead for frequent facet value requests.

## TIMEOUT AND RESILIENCE STRATEGIES

**Configurable Timeouts:** Default 5-second timeout with context cancellation ensures queries don't hang indefinitely while allowing sufficient time for complex aggregations.

**Graceful Degradation:** If user enrichment services are unavailable, the system returns UUIDs rather than failing completely, ensuring core functionality remains available.

**Circuit Breaker Patterns:** The system implements timeout and retry logic to handle temporary Elasticsearch cluster issues without cascading failures.

**Error Classification:** Different error types (parse errors, field validation, ES errors) are handled appropriately with specific error messages and status codes.

## BEST PRACTICES FOR OPTIMAL PERFORMANCE

- 1. Use Specific Queries:** Narrow your base query with the 'query' parameter to reduce the dataset size before aggregation. More specific queries execute faster.
- 2. Reasonable Limits:** Set appropriate limits (10-50 typically) rather than always requesting the maximum 100 values to reduce processing time and network usage.
- 3. Efficient Filters:** For autocomplete, use 2-3 character prefixes rather than single characters to balance responsiveness with result quality.
- 4. Cache Results:** Client applications should cache facet values when appropriate, especially for facets that change infrequently like user lists or project names.
- 5. Monitor Performance:** Use the execution time logging to identify slow queries and optimize accordingly by adjusting query complexity or limits.

## PERFORMANCE MONITORING

The system provides comprehensive logging of execution times, query complexity, and result sizes. Key metrics include parse time, ES query execution time, user resolution time, and total request processing time. These metrics enable identification of bottlenecks and optimization opportunities.

# Error Handling & Troubleshooting

## COMMON ERROR SCENARIOS AND RESOLUTIONS

1. UNSUPPORTED FACET ERROR
- Error: ErrUnsupportedFacet
- Cause: Requested facet not in SupportedFacets set
- Resolution: Use GetFacets endpoint to discover available facets
- Prevention: Validate facet names against supported list
2. QUERY PARSING ERRORS
- Error: ANTLR parsing failure
- Cause: Invalid query syntax (unbalanced parentheses, invalid operators)
- Resolution: Fix query syntax or use simpler query patterns
- Example: "status:open AND priority:" (missing value) should be "status:open"
3. FIELD VALIDATION ERRORS
- Error: Invalid field names in query
- Cause: Using non-existent or non-searchable fields
- Resolution: Verify field names against case document schema
- Prevention: Use field discovery tools to identify available fields
4. ELASTICSEARCH TIMEOUT ERRORS
- Error: Context deadline exceeded
- Cause: Query too complex or ES cluster under load
- Resolution: Simplify query, reduce limit, or increase timeout
- Optimization: Use more specific base queries to reduce dataset size
5. PERMISSION/ACCESS ERRORS
- Error: Project access denied or empty results
- Cause: User lacks access to requested projects/organizations
- Resolution: Verify user permissions and organization membership
- Note: Security filtering may result in empty results rather than errors
6. SERVICE DEPENDENCY FAILURES
- Error: User enrichment failures
- Cause: UserService, ProjectService unavailable
- Resolution: System returns UUIDs instead of names (graceful degradation)
- Monitoring: Check service health and dependency status

## ERROR RESPONSE FORMATS

Validation Errors (400 Bad Request):

```
{
  "error": "InvalidArgument",
  "message": "Unsupported facet: invalid_field",
  "details": {
    "supported_facets": ["assignee", "status", "priority", ...]
  }
}
```

Query Parse Errors (400 Bad Request):

```
{
  "error": "InvalidArgument",
  "message": "Failed to parse query: unexpected token at position 15",
  "details": {
    "query": "status:open AND",
    "position": 15
  }
}
```

Timeout Errors (504 Gateway Timeout):

```
{
  "error": "DeadlineExceeded",
  "message": "Query execution timed out after 5s",
  "details": {
    "timeout": "5s",
    "suggestion": "Try reducing query complexity or limit parameter"
  }
}
```

## DEBUGGING STRATEGIES

1. Enable Debug Logging: Set log level to DEBUG to see detailed query processing
2. Query Simplification: Start with simple queries and add complexity incrementally
3. Limit Testing: Use small limits (5-10) during development to reduce processing time
4. Field Verification: Use GetFacets to verify facet availability before using
5. Permission Checking: Verify organization and project access independently

## PERFORMANCE TROUBLESHOOTING

Slow Queries:

- Check base query complexity and specificity
- Monitor shard count and distribution
- Verify index health and performance metrics
- Consider query optimization or limit reduction

High Memory Usage:

- Review aggregation cardinality (unique value count)
- Check for inefficient wildcard patterns in filters
- Monitor fielddata usage on high-cardinality fields
- Consider aggregation circuit breakers

Network Issues:

- Monitor response payload sizes
- Check for oversized limit parameters
- Verify connection pooling effectiveness
- Review serialization/deserialization performance

## MONITORING AND ALERTING

Key metrics to monitor:

- Query execution time distribution
- Error rate by error type
- Memory usage during aggregations
- Network payload sizes
- User service dependency health

Set up alerts for:

- Query timeout rates > 5%
- Memory usage > 80% during aggregations
- User service failure rates > 1%
- Response time P95 > 2 seconds

# Practical Use Cases & Integration Patterns

## FRONTEND INTEGRATION PATTERNS

1. DROPDOWN FILTER POPULATION  
Scenario: User clicks on assignee filter dropdown  
Implementation:
  - Call SearchFacetValues with current search query as base filter
  - Use empty filter parameter to get all available assignees
  - Populate dropdown with enriched user names and emails
  - Cache results for 5 minutes to reduce server load
2. SEARCH-AS-YOU-TYPE AUTOCOMPLETE  
Scenario: User types in search field with facet suggestions  
Implementation:
  - Debounce user input (300ms delay)
  - Call SearchFacetValues with typed text as filter parameter
  - Display matching facet values as suggestions
  - Format suggestions as "field:value" for easy query construction
3. CONTEXTUAL FILTER DISCOVERY  
Scenario: Show relevant filters based on current search results  
Implementation:
  - Use current search query as base filter
  - Call SearchFacetValues for multiple facets in parallel
  - Display only facets with multiple available values
  - Hide facets where all results have same value

## DASHBOARD AND ANALYTICS INTEGRATION

1. DYNAMIC GROUPING OPTIONS  
Scenario: Dashboard needs available grouping dimensions  
Implementation:
  - Query each potential grouping facet
  - Show only facets with sufficient cardinality (>1 unique value)
  - Order by value count to prioritize high-cardinality facets
  - Provide user-friendly names through enrichment
2. FILTER VALIDATION  
Scenario: Validate saved searches and dashboard filters  
Implementation:
  - Check if filter values still exist in current dataset
  - Remove obsolete filter values from saved configurations
  - Suggest alternative values when original values are unavailable

## ADMINISTRATIVE AND REPORTING USE CASES

1. DATA QUALITY MONITORING  
Scenario: Monitor data completeness and quality  
Implementation:
  - Query facets to identify missing or invalid values
  - Check for expected vs actual facet value distributions
  - Alert on unusual patterns (e.g., too many "unknown" values)
2. USER ACTIVITY ANALYSIS  
Scenario: Understand system usage patterns  
Implementation:
  - Analyze assignee facet values to identify active users
  - Monitor project facet distributions for resource allocation
  - Track service/team facet usage for organizational insights

## ADVANCED INTEGRATION PATTERNS

1. CASCADING FILTERS  
Implementation:
  - Use parent filter selection as query parameter for child facets
  - Example: Select project first, then show only assignees from that project
  - Maintain filter dependency chain for intuitive user experience
2. BULK OPERATIONS SUPPORT  
Implementation:
  - Use facet values to enable bulk selection ("all high priority cases")
  - Provide value count information for impact assessment
  - Enable bulk actions based on facet value selections
3. EXTERNAL SYSTEM INTEGRATION  
Implementation:
  - Use service/team facets for external ITSM system mapping
  - Correlate facet values with external monitoring systems
  - Export facet distributions for business intelligence tools

## CLIENT-SIDE OPTIMIZATION STRATEGIES

1. Request Batching:
  - Combine multiple facet requests into parallel calls
  - Use Promise.all() or similar for concurrent execution
  - Implement request deduplication for identical queries
2. Response Caching:
  - Cache stable facets (users, projects) longer than dynamic ones (status)
  - Use query hash as cache key to enable query-specific caching
  - Implement cache invalidation on relevant data changes
3. Progressive Loading:
  - Load high-priority facets first (status, priority)
  - Lazy-load secondary facets (custom attributes, teams)
  - Show loading states and partial results during fetch

## ERROR HANDLING IN CLIENT APPLICATIONS

1. Graceful Degradation:
  - Fall back to cached values when service is unavailable
  - Provide manual text entry when facet loading fails
  - Show error states with retry options
2. User Experience Optimization:
  - Show loading indicators during facet value fetching
  - Provide search within results for high-cardinality facets
  - Implement keyboard navigation for autocomplete suggestions

## PERFORMANCE MONITORING IN PRODUCTION

1. Client Metrics:
  - Track facet load times and success rates
  - Monitor cache hit rates and effectiveness
  - Measure user interaction patterns with facet values
2. Server Monitoring:
  - Alert on slow facet queries (>2s response time)
  - Track facet value cardinality changes over time
  - Monitor memory usage during high-cardinality aggregations

# Appendix: Complete Request/Response Examples

## EXAMPLE 1: BASIC ASSIGNEE FACET REQUEST

```
Request:
POST /api/v1/cases/search-facet-values
{
  "org_id": 12345,
  "query": "status:open AND priority:high",
  "facet": "assignee",
  "filter": "john",
  "limit": 10,
  "work_type": "CASE"
}

Generated Elasticsearch Query:
{
  "query": {
    "bool": {
      "must": [
        {"term": {"status": 1}},
        {"term": {"priority": 3}},
        {"term": {"org_id": 12345}},
        {"terms": {"project_id": ["proj-1", "proj-2", "proj-3"]}}
      ]
    },
    "size": 0,
    "aggs": {
      "assignee": {
        "terms": {
          "field": "assignee_id",
          "size": 10,
          "shard_size": 20,
          "execution_hint": "map",
          "include": ".*john.*"
        }
      }
    }
  }
}

Response:
{
  "field_name": "assignee",
  "values": [
    {
      "field_value": "john.doe@datadog.com",
      "count": 15
    },
    {
      "field_value": "johnny.smith@datadog.com",
      "count": 8
    },
    {
      "field_value": "john.wilson@datadog.com",
      "count": 3
    }
  ]
}
```

## EXAMPLE 2: CUSTOM ATTRIBUTE FACET REQUEST

```
Request:
{
  "org_id": 12345,
  "query": "created_at:[now-30d TO now]",
  "facet": "custom_attributes.environment",
  "filter": "prod",
  "limit": 5
}

Generated Elasticsearch Query (Complex Nested):
{
  "query": {
    "bool": {
      "must": [
        {
          "range": {
            "created_at": {
              "gte": "2023-11-01T00:00:00Z",
              "lte": "2023-12-01T00:00:00Z"
            }
          }
        },
        {"term": {"org_id": 12345}}
      ]
    },
    "size": 0,
    "aggs": {
      "custom_attributes.environment": {
        "nested": {
          "path": "custom_attributes"
        },
        "aggs": {
          "filter_custom_attr": {
            "filter": {
              "term": {"custom_attributes.key": "environment"}
            },
            "aggs": {
              "text_values": {
                "terms": {
                  "field": "custom_attributes.value_text.keyword",
                  "size": 5,
                  "include": ".*prod.*"
                }
              }
            }
          }
        }
      }
    }
  }
}

Response:
{
  "field_name": "custom_attributes.environment",
  "values": [
    {"field_value": "production", "count": 45},
    {"field_value": "prod-eu", "count": 23},
    {"field_value": "staging-prod", "count": 12}
  ]
}
```

## EXAMPLE 3: STATUS FACET WITHOUT FILTER

```
Request:
{
  "org_id": 12345,
  "query": "",
  "facet": "status",
  "filter": "",
  "limit": 20
}

Response:
{
  "field_name": "status",
  "values": [
    {"field_value": "OPEN", "count": 234},
    {"field_value": "IN PROGRESS", "count": 156},
    {"field_value": "CLOSED", "count": 89},
    {"field_value": "ARCHIVED", "count": 23}
  ]
}
```