MCP Documentation

Ayush, Pritam, Kaustubh

Wazuh MCP Server - Complete Pipeline Documentation

Table of Contents

Overview

The Wazuh MCP Server provides three distinct query pipelines, each optimized for different use cases:

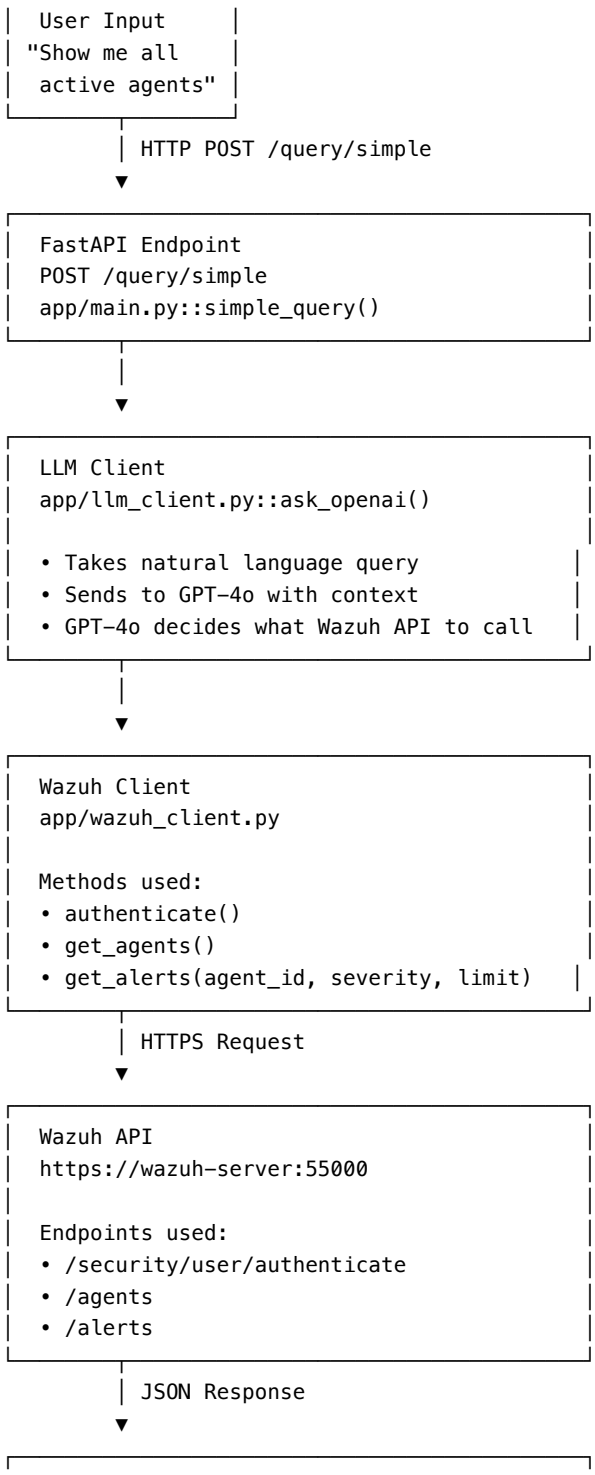
Pipeline	Complexity	Speed	Best For	Requires Indexer
Simple NL	Low	~2s	Basic queries	No
Advanced NL + DSL	High	~3s	Complex searches	Yes
Pre-built DSL	Medium	<1s	Automation	Yes

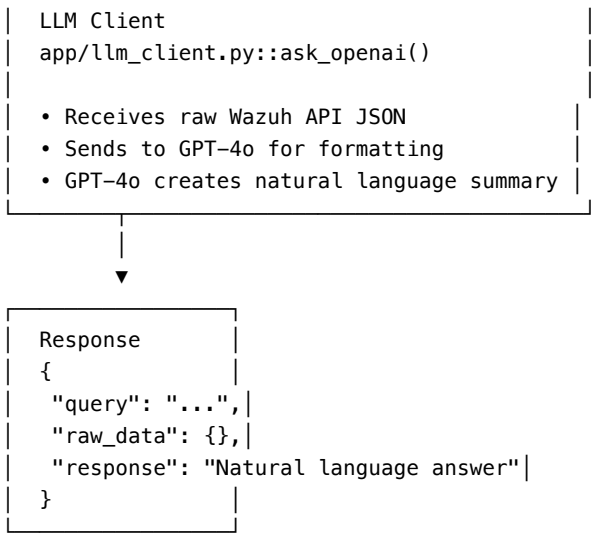
Pipeline 1: Simple Natural Language

Overview

Direct communication with Wazuh API using GPT-4o for query understanding and response formatting. No DSL generation or OpenSearch involved.

Architecture Flow


User Input
"Show me all active agents"
HTTP POST /query/simple
FastAPI Endpoint
POST /query/simple
app/main.py::simple_query()
LLM Client
app/llm_client.py::ask_openai()
• Takes natural language query
• Sends to GPT-4o with context
• GPT-4o decides what Wazuh API to call
Wazuh Client
app/wazuh_client.py
Methods used:
• authenticate()
• get_agents()
• get_alerts(agent_id, severity, limit)
HTTPS Request
Wazuh API
https://wazuh-server:55000
Endpoints used:
• /security/user/authenticate
• /agents
• /alerts
JSON Response



Code Flow Detail

1. Request Entry Point

```python
# File: app/main.py
@app.post("/query/simple")
async def simple_query(request: NaturalLanguageQuery):
    """
    Handles simple natural language queries via Wazuh API only.
    No DSL building, no OpenSearch indexer needed.
    """
    try:
        # Step 1: Send query to OpenAI with Wazuh context
        # Step 2: OpenAI determines which Wazuh API to call
        # Step 3: Fetch data from Wazuh API
        # Step 4: Send results back to OpenAI for formatting
        # Step 5: Return natural language response
```

2. LLM Processing

```python
# File: app/llm_client.py
async def ask_openai(query: str, wazuh_client: WazuhClient) -> Dict:
    """
    Uses GPT-4o to:
    1. Understand the user's question
    2. Determine which Wazuh API endpoint to call
    3. Make the API call
    4. Format the response in natural language
    """
```

```
    # OpenAI system prompt includes:
    # - Available Wazuh API functions
    # - How to call get_agents(), get_alerts()
    # - Response formatting guidelines

    response = openai.chat.completions.create(
        model="gpt-4o",
        messages=[
            {"role": "system", "content": WAZUH_CONTEXT},
            {"role": "user", "content": query}
        ],
        functions=[
            # Function definitions for get_agents, get_alerts
        ],
        function_call="auto"
    )

    # If GPT-4o calls a function:
    # - Execute the Wazuh API call
    # - Send results back to GPT-4o
    # - Get natural language summary
```

3. Wazuh API Interaction

```
# File: app/wazuh_client.py
class WazuhClient:
    async def get_agents(self) -> Dict:
        """Fetch all agents from Wazuh API"""
        response = await self.client.get(
            f"{self.base_url}/agents",
            headers={"Authorization": f"Bearer {self.token}"}
        )
        return response.json()

    async def get_alerts(self, agent_id=None, severity=None, limit=10):
        """Fetch alerts with optional filters"""
        params = {"limit": limit}
        if agent_id:
            params["agent_id"] = agent_id
        if severity:
            params["rule.level"] = f"gte:{severity}"

        response = await self.client.get(
            f"{self.base_url}/alerts",
            params=params,
```

```
            headers={"Authorization": f"Bearer {self.token}"}
        )
        return response.json()
```

Use Cases

- "Show me all active agents"
- "List agents by status"
- "What agents are disconnected?"
- "Show me recent alerts"
- Complex time-based queries (use Pipeline 2)
- Aggregations (use Pipeline 2)

Advantages

- No OpenSearch/Indexer required
- Simple and fast for basic queries
- Good for beginners
- Lower latency for agent queries

Limitations

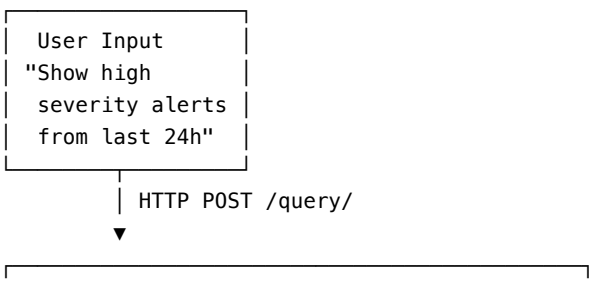
- Limited to Wazuh API capabilities
- No complex filtering
- No time range queries
- No aggregations

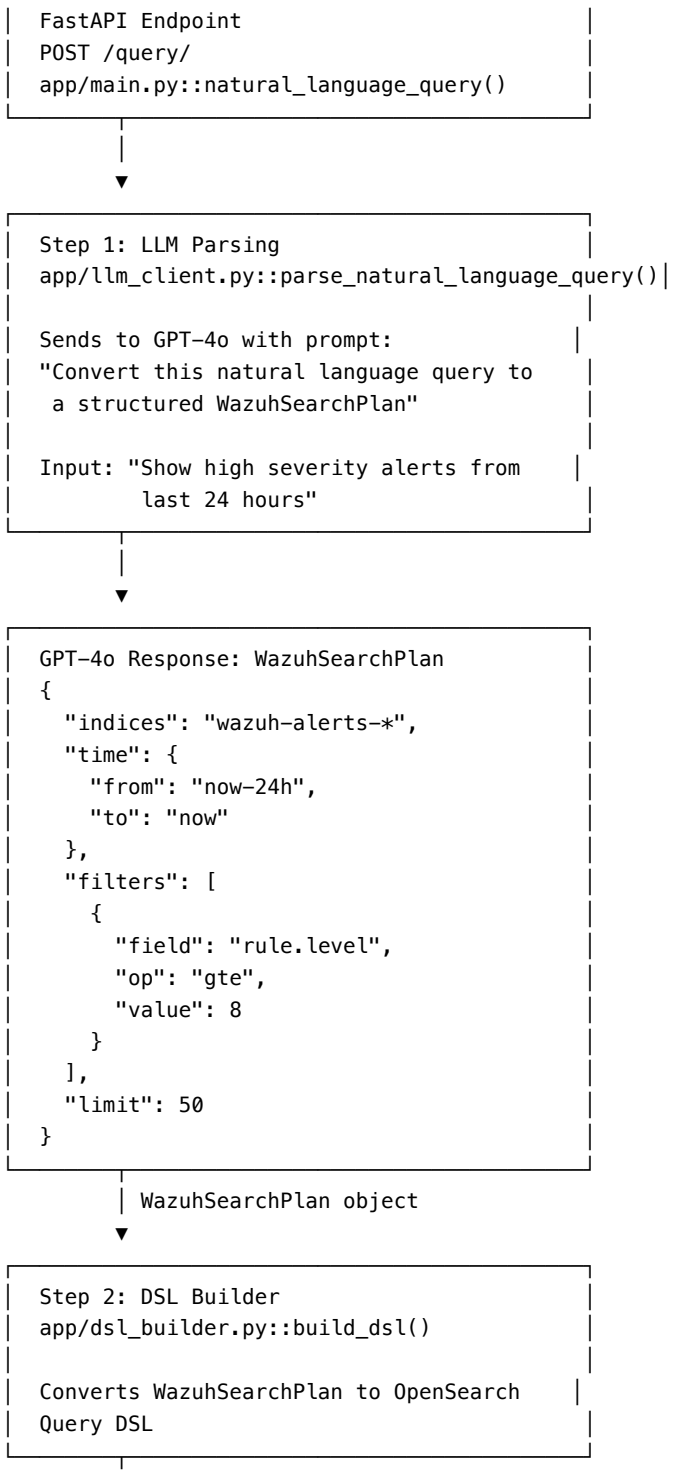
Pipeline 2: Advanced Natural Language with DSL

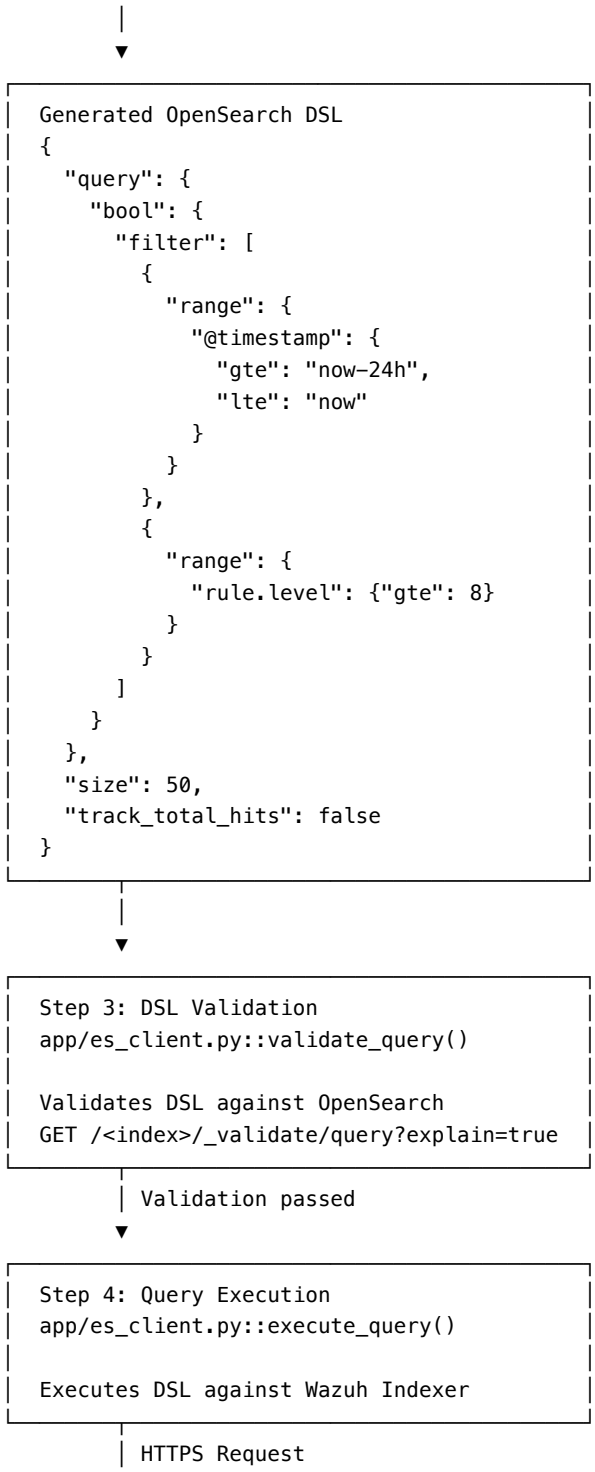
Overview

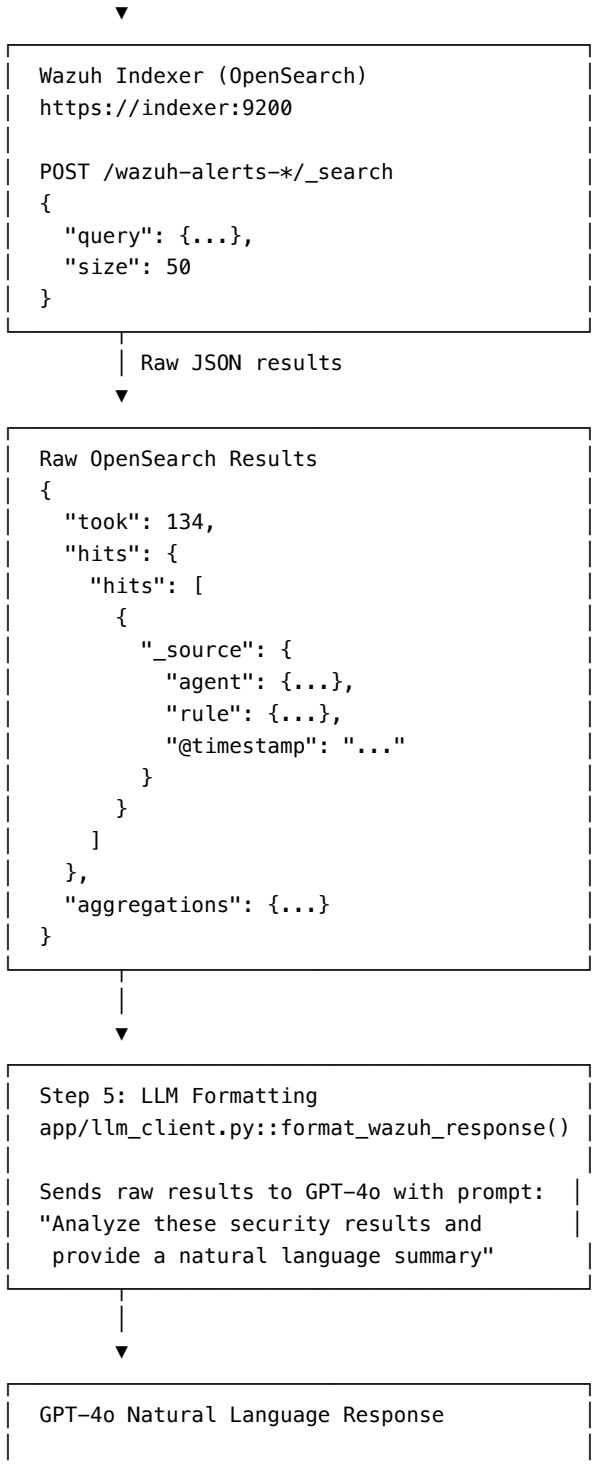
Complete pipeline that converts natural language to OpenSearch DSL, executes against Wazuh Indexer, and formats results back to natural language using GPT-4o.

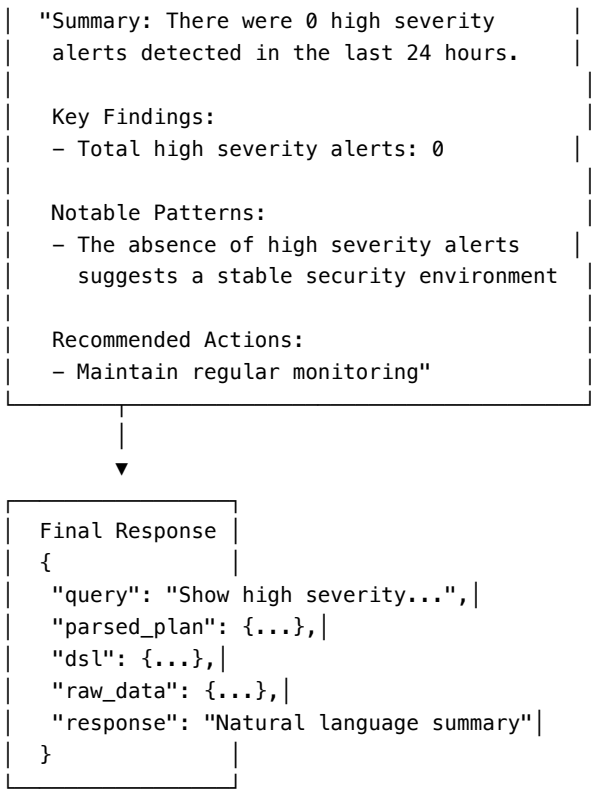
Architecture Flow




FastAPI Endpoint
POST /query/
app/main.py::natural_language_query()
Step 1: LLM Parsing
app/llm_client.py::parse_natural_language_query()
Sends to GPT-4o with prompt:
"Convert this natural language query to
a structured WazuhSearchPlan"
Input: "Show high severity alerts from
last 24 hours"
GPT-4o Response: WazuhSearchPlan
{
"indices": "wazuh-alerts-*",
"time": {
"from": "now-24h",
"to": "now"
},
"filters": [
{
"field": "rule.level",
"op": "gte",
"value": 8
}
],
"limit": 50
}
WazuhSearchPlan object
Step 2: DSL Builder
app/dsl_builder.py::build_dsl()
Converts WazuhSearchPlan to OpenSearch
Query DSL


Generated OpenSearch DSL
{
"query": {
"bool": {
"filter": [
{
"range": {
"@timestamp": {
"gte": "now-24h",
"lte": "now"
}
}
},
{
"range": {
"rule.level": {"gte": 8}
}
}
]
}
},
"size": 50,
"track_total_hits": false
}
Step 3: DSL Validation
app/es_client.py::validate_query()
Validates DSL against OpenSearch
GET /<index>/_validate/query?explain=true
Validation passed
Step 4: Query Execution
app/es_client.py::execute_query()
Executes DSL against Wazuh Indexer
HTTPS Request


Wazuh Indexer (OpenSearch)
https://indexer:9200
POST /wazuh-alerts-*/_search
{
"query": {...},
"size": 50
}
Raw JSON results
Raw OpenSearch Results
{
"took": 134,
"hits": {
"hits": [
{
"_source": {
"agent": {...},
"rule": {...},
"@timestamp": "..."
}
}
]
},
"aggregations": {...}
}
Step 5: LLM Formatting
app/llm_client.py::format_wazuh_response()
Sends raw results to GPT-4o with prompt:
"Analyze these security results and
provide a natural language summary"
GPT-4o Natural Language Response



Code Flow Detail

1. Request Entry Point

```python
# File: app/main.py
@app.post("/query/")
async def natural_language_query(request: NaturalLanguageQuery):
    """
    Complete pipeline: NL → Parse → DSL → Indexer → Format → NL
    """
    # Step 1: Parse natural language to WazuhSearchPlan
    plan = await parse_natural_language_query(request.query)

    # Step 2: Build OpenSearch DSL from plan
    dsl = build_dsl(plan)

    # Step 3: Validate DSL
    validation = validate_query(plan.indices, dsl)
    if not validation["valid"]:
        raise HTTPException(400, f"Invalid query: {validation['error']}")
```

```
    # Step 4: Execute query against Wazuh Indexer
    results = execute_query(plan.indices, dsl)

    # Step 5: Format results with GPT-4o
    formatted_response = await format_wazuh_response(
        request.query, results
    )

    return {
        "query": request.query,
        "parsed_plan": plan.dict(),
        "dsl": dsl,
        "raw_data": results,
        "response": formatted_response
    }
```

2. Natural Language Parsing

```
# File: app/llm_client.py
async def parse_natural_language_query(query: str) -> WazuhSearchPlan:
    """
    Uses GPT-4o to convert natural language to structured WazuhSearchPlan
    """

    system_prompt = """
    You are a Wazuh SIEM query parser. Convert natural language queries
    into structured WazuhSearchPlan JSON.

    Available fields:
    - agent.name, agent.id, agent.ip
    - rule.level (0-15), rule.description
    - @timestamp
    - data.vulnerability.severity

    Time expressions:
    - "last 24 hours" → "now-24h"
    - "last 7 days" → "now-7d"
    - "yesterday" → "now-1d/d"

    Operators: eq, ne, gt, gte, lt, lte, contains

    Output only valid JSON matching WazuhSearchPlan schema.
    """

    response = await openai.chat.completions.create(
```

```
        model="gpt-4o",
        messages=[
            {"role": "system", "content": system_prompt},
            {"role": "user", "content": query}
        ],
        response_format={"type": "json_object"}
    )

    # Parse JSON response into WazuhSearchPlan Pydantic model
    plan_dict = json.loads(response.choices[0].message.content)
    return WazuhSearchPlan(**plan_dict)
```

3. DSL Builder

```
# File: app/dsl_builder.py
def build_dsl(plan: WazuhSearchPlan) -> Dict:
    """
    Converts WazuhSearchPlan to OpenSearch Query DSL
    """
    query = {"bool": {"filter": []}}

    # Add time range filter
    if plan.time:
        query["bool"]["filter"].append({
            "range": {
                "@timestamp": {
                    "gte": plan.time.from_time,
                    "lte": plan.time.to
                }
            }
        })

    # Add field filters
    for filter_item in plan.filters:
        clause = filter_to_clause(filter_item)
        query["bool"]["filter"].append(clause)

    # Add must_not filters
    if plan.must_not:
        query["bool"]["must_not"] = [
            filter_to_clause(f) for f in plan.must_not
        ]

    # Add query_string if present
    if plan.query_string:
        query["bool"]["must"] = [
```

```
            {"query_string": {"query": plan.query_string}}
        ]

    dsl = {
        "query": query,
        "size": 0 if plan.aggregation else plan.limit,
        "track_total_hits": False
    }

    # Add aggregation if present
    if plan.aggregation:
        dsl["aggs"] = build_aggregation(plan.aggregation)

    return dsl

def filter_to_clause(filter_item: FilterItem) -> Dict:
    """Converts FilterItem to OpenSearch filter clause"""
    field = filter_item.field
    op = filter_item.op
    value = filter_item.value

    if op == "eq":
        return {"term": {field: value}}
    elif op in ["gt", "gte", "lt", "lte"]:
        return {"range": {field: {op: value}}}
    elif op == "contains":
        return {"wildcard": {field: f"*{value}*"}}
    elif op == "ne":
        return {"bool": {"must_not": [{"term": {field: value}}]}}
```

4. Query Validation

```
# File: app/es_client.py
def validate_query(indices: str, dsl: Dict) -> Dict:
    """Validates OpenSearch DSL before execution"""
    client = get_client()

    try:
        response = client.transport.perform_request(
            "GET",
            f"/{indices}/_validate/query",
            params={"explain": "true"},
            body={"query": dsl["query"]}
        )

        return {
```

```
            "valid": response.get("valid", False),
            "error": None if response.get("valid") else response
        }
    except Exception as e:
        return {"valid": False, "error": str(e)}
```

5. Query Execution

```
# File: app/es_client.py
def execute_query(indices: str, dsl: Dict) -> Dict:
    """Executes OpenSearch DSL query"""
    client = get_client()

    try:
        response = client.search(
            index=indices,
            body=dsl
        )
        return response
    except Exception as e:
        raise Exception(f"Query execution failed: {str(e)}")
```

6. Response Formatting

```
# File: app/llm_client.py
async def format_wazuh_response(query: str, raw_data: Dict) -> str:
    """
    Uses GPT-4o to format raw OpenSearch results into natural language
    """

    system_prompt = """
    You are a Wazuh SIEM security analyst. Analyze the query results
    and provide a clear, actionable summary in natural language.

    Structure your response as:
    1. Summary: Brief overview
    2. Key Findings: Important data points with numbers
    3. Notable Patterns or Concerns: Security insights
    4. Recommended Actions: What the operator should do
    """

    user_prompt = f"""
    Original Query: {query}

    Raw Results:
    {json.dumps(raw_data, indent=2)}
```

```
Provide a natural language analysis.
"""

response = await openai.chat.completions.create(
    model="gpt-4o",
    messages=[
        {"role": "system", "content": system_prompt},
        {"role": "user", "content": user_prompt}
    ]
)

return response.choices[0].message.content
```

Use Cases

- "Show high severity alerts from last 24 hours"
- "Count alerts by rule level in last 7 days"
- "Find authentication failures from last week"
- "Show alerts from agent DC01"
- "Alert trends over time"
- Complex filtering and aggregations

Advantages

- Full OpenSearch DSL capabilities
- Complex time-based queries
- Aggregations and analytics
- Natural language input and output
- Detailed security insights from GPT-4o

Limitations

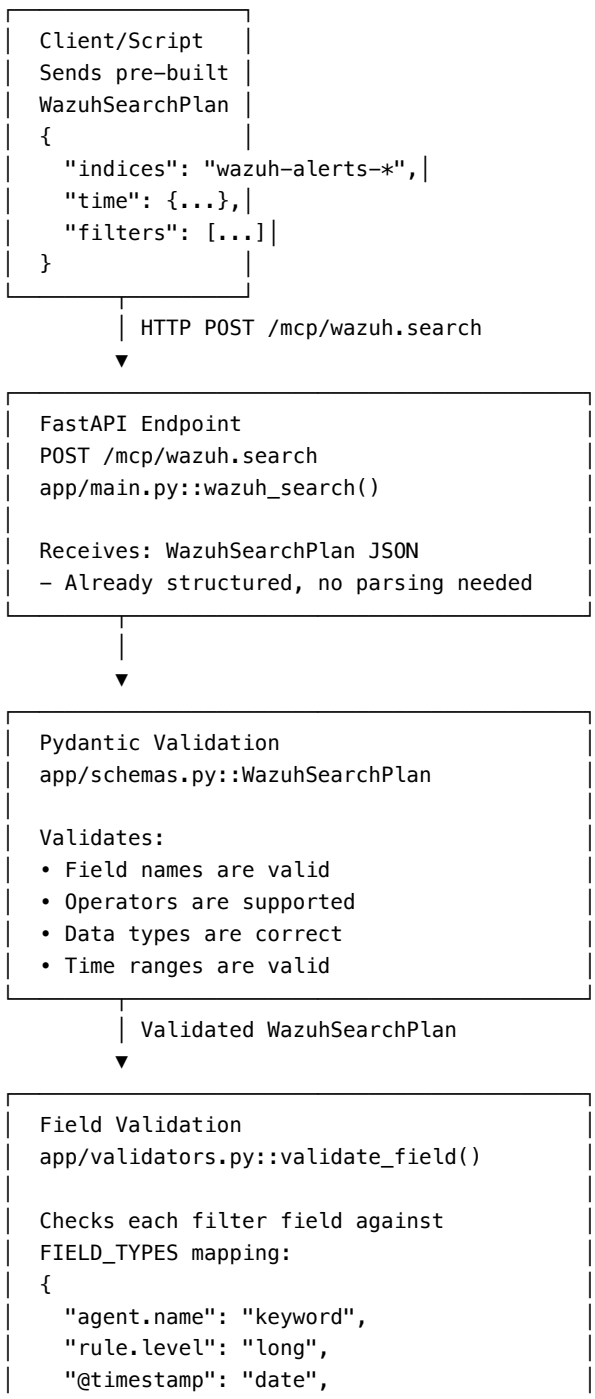
- Requires Wazuh Indexer
- Slower due to multiple LLM calls (~3s)
- More complex error handling

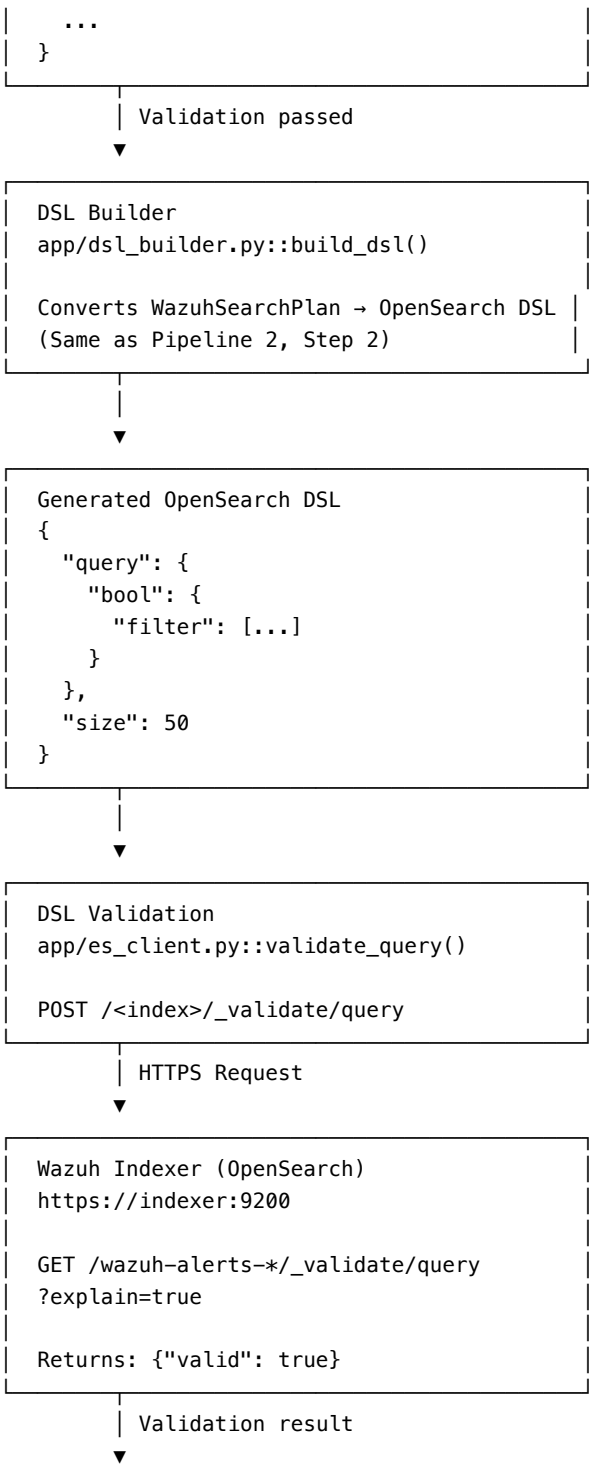
Pipeline 3: Pre-built DSL Queries

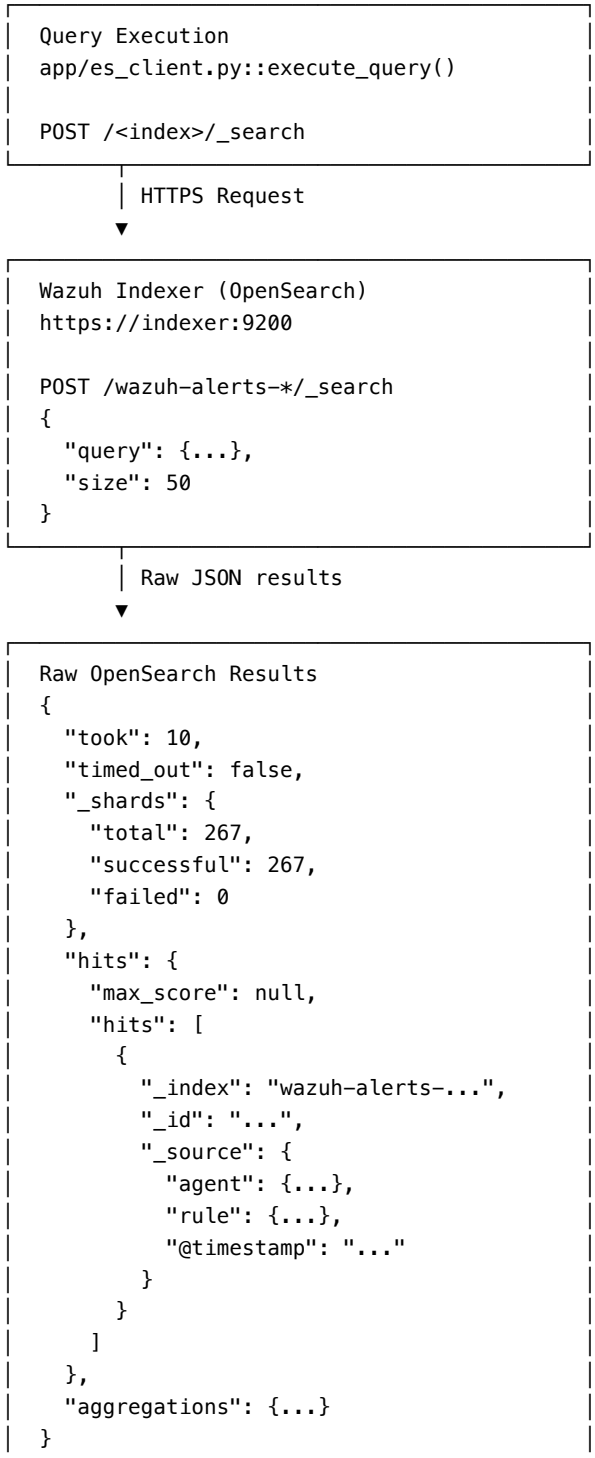
Overview

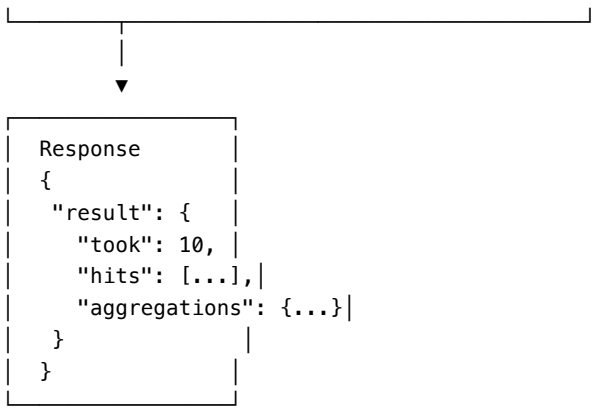
Direct DSL query submission for programmatic access and automation. No natural language processing - pure OpenSearch DSL in, raw JSON out.

Architecture Flow




...
}
Validation passed
DSL Builder
app/dsl_builder.py::build_dsl()
Converts WazuhSearchPlan → OpenSearch DSL
(Same as Pipeline 2, Step 2)
Generated OpenSearch DSL
{
"query": {
"bool": {
"filter": [...]
}
},
"size": 50
}
DSL Validation
app/es_client.py::validate_query()
POST /<index>/_validate/query
HTTPS Request
Wazuh Indexer (OpenSearch)
https://indexer:9200
GET /wazuh-alerts-*/_validate/query
?explain=true
Returns: {"valid": true}
Validation result


Query Execution
app/es_client.py::execute_query()
POST /<index>/_search
HTTPS Request
Wazuh Indexer (OpenSearch)
https://indexer:9200
POST /wazuh-alerts-*/_search
{
"query": {...},
"size": 50
}
Raw JSON results
Raw OpenSearch Results
{
"took": 10,
"timed_out": false,
"_shards": {
"total": 267,
"successful": 267,
"failed": 0
},
"hits": {
"max_score": null,
"hits": [
{
"_index": "wazuh-alerts-...",
"_id": "...",
"_source": {
"agent": {...},
"rule": {...},
"@timestamp": "..."
}
}
]
},
"aggregations": {...}
}



Code Flow Detail

1. Request Entry Point

```python
# File: app/main.py
@app.post("/mcp/wazuh.search")
async def wazuh_search(plan: WazuhSearchPlan):
    """
    Direct DSL query execution. No LLM involved.
    Fast path for automation and programmatic access.
    """
    # Step 1: Pydantic validation (automatic)
    # Step 2: Field validation
    # Step 3: Build DSL
    # Step 4: Validate DSL
    # Step 5: Execute query
    # Step 6: Return raw results

    # Build DSL from plan
    dsl = build_dsl(plan)

    # Validate query
    validation = validate_query(plan.indices, dsl)
    if not validation["valid"]:
        raise HTTPException(
            status_code=400,
            detail=f"validate failed: {validation['error']}"
        )

    # Execute query
    try:
        results = execute_query(plan.indices, dsl)
```

```
        return {"result": results}
    except Exception as e:
        raise HTTPException(
            status_code=500,
            detail=f"Query execution failed: {str(e)}"
        )
```

2. Pydantic Schema Validation

```
# File: app/schemas.py
class TimeRange(BaseModel):
    from_time: str = Field(alias="from")
    to: str = "now"
    timezone: str = "UTC"

class OpEnum(str, Enum):
    eq = "eq"
    ne = "ne"
    gt = "gt"
    gte = "gte"
    lt = "lt"
    lte = "lte"
    contains = "contains"

class FilterItem(BaseModel):
    field: str
    op: OpEnum
    value: Union[str, int, float, bool]

class AggregationType(str, Enum):
    terms = "terms"
    date_histogram = "date_histogram"
    avg = "avg"
    sum = "sum"
    max = "max"
    min = "min"

class Aggregation(BaseModel):
    type: AggregationType
    field: str
    size: Optional[int] = 10
    interval: Optional[str] = None

class WazuhSearchPlan(BaseModel):
    model_config = {"populate_by_name": True}
```

```
    indices: str = "wazuh-alerts-*"
    time: Optional[TimeRange] = None
    filters: List[FilterItem] = []
    must_not: List[FilterItem] = []
    query_string: Optional[str] = None
    aggregation: Optional[Aggregation] = None
    limit: int = 50
    dry_run: bool = False
```

3. Field Validation

```
# File: app/validators.py
FIELD_TYPES = {
    # Agent fields
    "agent.id": "keyword",
    "agent.name": "keyword",
    "agent.ip": "ip",

    # Rule fields
    "rule.id": "keyword",
    "rule.level": "long",
    "rule.description": "text",

    # Manager fields
    "manager.name": "keyword",

    # Vulnerability fields
    "data.vulnerability.severity": "keyword",

    # Timestamp
    "@timestamp": "date"
}

def validate_field(field: str) -> bool:
    """Validates if field exists in Wazuh indices"""
    return field in FIELD_TYPES

def validate_operator(field: str, op: str, value: Any) -> bool:
    """Validates if operator is compatible with field type"""
    field_type = FIELD_TYPES.get(field)

    if field_type == "keyword":
        return op in ["eq", "ne", "contains"]
    elif field_type in ["long", "integer"]:
        return op in ["eq", "ne", "gt", "gte", "lt", "lte"]
    elif field_type == "date":
```

```
        return op in ["gt", "gte", "lt", "lte"]

    return False
```

Use Cases

- Automated monitoring scripts
- Integration with other tools
- Scheduled reports
- Custom dashboards
- API-to-API communication
- Testing and development

Advantages

- Fastest response time (<1s)
- No LLM overhead
- Predictable behavior
- Full OpenSearch DSL control
- Easy to test and debug
- Perfect for automation

Limitations

- Requires OpenSearch DSL knowledge
- No natural language support
- Raw JSON output (not human-friendly)
- Manual query construction

Component Details

WazuhClient (app/wazuh_client.py)

```python
class WazuhClient:
    """Async HTTP client for Wazuh API"""

    def __init__(self, base_url: str, username: str, password: str):
        self.base_url = base_url
        self.username = username
        self.password = password
        self.client = httpx.AsyncClient(verify=False, timeout=30.0)
        self.token = None

    async def authenticate(self) -> str:
        """Get JWT token from Wazuh API"""
        response = await self.client.get(
```

```
            f"{self.base_url}/security/user/authenticate",
            auth=(self.username, self.password)
        )
        self.token = response.json()["data"]["token"]
        return self.token

    async def get_agents(self) -> Dict:
        """Fetch all agents"""
        # GET /agents

    async def get_alerts(self, agent_id=None, severity=None, limit=10):
        """Fetch alerts with filters"""
        # GET /alerts?agent_id=...&rule.level=gte:8

    async def restart_manager(self) -> Dict:
        """Restart Wazuh manager"""
        # PUT /manager/restart

    async def check_api(self) -> bool:
        """Health check for Wazuh API"""
        # GET /

    async def close(self):
        """Close HTTP client"""
        await self.client.aclose()
```

LLM Client (app/llm_client.py)

```
# GPT-4o Functions for all pipelines

async def parse_natural_language_query(query: str) -> WazuhSearchPlan:
    """Pipeline 2: Convert NL to WazuhSearchPlan"""
    # Uses GPT-4o with structured output
    # Returns validated Pydantic model

async def format_wazuh_response(query: str, raw_data: Dict) -> str:
    """Pipeline 2: Convert raw results to NL"""
    # Uses GPT-4o with security analyst prompt
    # Returns natural language summary

async def ask_openai(query: str, wazuh_client: WazuhClient) -> Dict:
    """Pipeline 1: Complete NL query via Wazuh API"""
    # Uses GPT-4o with function calling
    # Calls Wazuh API functions directly
    # Returns formatted response
```

DSL Builder (app/dsl_builder.py)

```python
def build_dsl(plan: WazuhSearchPlan) -> Dict:
    """Convert WazuhSearchPlan to OpenSearch DSL"""
    # Used by Pipeline 2 and 3
    # Handles:
    # - Time range filters
    # - Field filters (eq, ne, gt, gte, lt, lte, contains)
    # - Must not filters
    # - Query strings
    # - Aggregations (terms, date_histogram, metrics)

def filter_to_clause(filter_item: FilterItem) -> Dict:
    """Convert single filter to OpenSearch clause"""
    # Maps operators to OpenSearch query types:
    # - eq → term
    # - ne → bool.must_not.term
    # - gt/gte/lt/lte → range
    # - contains → wildcard

def build_aggregation(agg: Aggregation) -> Dict:
    """Build aggregation clause"""
    # Supports:
    # - terms: Count by field
    # - date_histogram: Time-based grouping
    # - avg/sum/max/min: Metric aggregations
```

OpenSearch Client (app/es_client.py)

```python
def get_client() -> OpenSearch:
    """Create OpenSearch client"""
    # SSL configuration
    # Authentication
    # Connection pooling

def validate_query(indices: str, dsl: Dict) -> Dict:
    """Validate DSL before execution"""
    # GET /<index>/_validate/query?explain=true
    # Returns: {"valid": bool, "error": str}

def execute_query(indices: str, dsl: Dict) -> Dict:
    """Execute OpenSearch query"""
    # POST /<index>/_search
    # Returns raw OpenSearch response
```

Examples

Pipeline 1: Simple Natural Language

Request:

```
curl -X POST http://localhost:8000/query/simple \
  -H "Content-Type: application/json" \
  -d '{
    "query": "Show me all active agents"
  }'
```

Response:


```
{
  "query": "Show me all active agents",
  "raw_data": {
    "total": 4,
    "agents": [
      {
        "id": "000",
        "name": "hpotwaz",
        "status": "active",
        "ip": "127.0.0.1"
      }
    ]
  },
  "response": "Summary: You have 4 agents registered. Only 1 agent (hpotwaz) is currently active. The
}
```


Pipeline 2: Advanced Natural Language with DSL

Request:

```
curl -X POST http://localhost:8000/query/ \
  -H "Content-Type: application/json" \
  -d '{
    "query": "Show high severity alerts from last 24 hours"
  }'
```

Response:


```
{
  "query": "Show high severity alerts from last 24 hours",
  "parsed_plan": {
    "indices": "wazuh-alerts-*",
    "time": {
      "from": "now-24h",
      "to": "now"
    },
```

```
    "filters": [
      {
        "field": "rule.level",
        "op": "gte",
        "value": 8
      }
    ],
    "limit": 50
  },
  "dsl": {
    "query": {
      "bool": {
        "filter": [
          {
            "range": {
              "@timestamp": {
                "gte": "now-24h",
                "lte": "now"
              }
            }
          },
          {
            "range": {
              "rule.level": {
                "gte": 8
              }
            }
          }
        ]
      }
    },
    "size": 50,
    "track_total_hits": false
  },
  "raw_data": {
    "took": 134,
    "hits": {
      "total": {"value": 0},
      "hits": []
    }
  },
  "response": "Summary: There were no high severity alerts (level ≥8) detected in the last 24 hours.\
}
```

Pipeline 3: Pre-built DSL

Request:

```
curl -X POST http://localhost:8000/mcp/wazuh.search \
  -H "Content-Type: application/json" \
  -d '{
    "indices": "wazuh-alerts-*",
    "time": {
      "from": "now-7d",
      "to": "now"
    },
    "filters": [
      {
        "field": "agent.name",
        "op": "eq",
        "value": "hpotwaz"
      }
    ],
    "limit": 10
  }'
```

Response:


```
{
  "result": {
    "took": 10,
    "timed_out": false,
    "_shards": {
      "total": 267,
      "successful": 267,
      "skipped": 266,
      "failed": 0
    },
    "hits": {
      "max_score": null,
      "hits": [
        {
          "_index": "wazuh-alerts-4.x-2025.12.09",
          "_id": "abc123",
          "_source": {
            "agent": {
              "name": "hpotwaz",
              "id": "000",
              "ip": "127.0.0.1"
            },
            "rule": {
```

```
            "level": 3,
            "id": "1002",
            "description": "Unknown problem somewhere in the system"
          },
          "@timestamp": "2025-12-09T10:30:00.000Z"
        }
      }
    ]
  }
 }
}
```

Aggregation Example:

```
curl -X POST http://localhost:8000/mcp/wazuh.search \
  -H "Content-Type: application/json" \
  -d '{
    "indices": "wazuh-alerts-*",
    "time": {
      "from": "now-7d",
      "to": "now"
    },
    "aggregation": {
      "type": "terms",
      "field": "rule.level",
      "size": 10
    }
  }'
```

Aggregation Response:

```
{
  "result": {
    "took": 131,
    "hits": {
      "total": {"value": 0},
      "hits": []
    },
    "aggregations": {
      "top_terms": {
        "buckets": [
          {
            "key": 3,
            "doc_count": 1245
          },
          {
            "key": 5,
            "doc_count": 89
```

```
        },
        {
          "key": 7,
          "doc_count": 12
        }
      ]
    }
  }
}
```

Performance Comparison

Metric	Pipeline 1	Pipeline 2	Pipeline 3
Avg Response Time	~2s	~3s	<1s
LLM Calls	1-2	2	0
OpenSearch Queries	0	1	1
Wazuh API Calls	1-2	0	0
Data Indexed	No	Yes	Yes
Best For	Basic queries	Complex analysis	Automation

Summary

When to Use Each Pipeline

Pipeline 1 (Simple NL): - Quick agent status checks - Basic operational queries - Users unfamiliar with SIEM concepts - No Indexer available - Not for: Complex searches, time-based queries, aggregations

Pipeline 2 (Advanced NL + DSL): - Complex security investigations - Time-based analysis - Trend detection - Natural language reports - Users want insights, not raw data - Not for: Real-time alerts, automation

Pipeline 3 (Pre-built DSL): - Automated monitoring - Scheduled reports - Integration with other tools - Performance-critical queries - Developers and power users - Not for: Non-technical users, ad-hoc queries

---

All three pipelines work together to provide flexibility for different use cases and user skill levels.