# OPMIG-06: Processing, Parsing & Transformation

> **Series:** OPMIG | **Notebook:** 6 of 9 | **Created:** December 2025

> **OpenPipeline Migration Series** | Notebook 6 of 9
> **Level:** Intermediate to Advanced
> **Estimated Time:** 75 minutes

## Learning Objectives

By completing this notebook, you will:

1. Master Dynatrace Pattern Language (DPL) for log parsing
2. Configure DQL processors for data transformation
3. ⭐ **NEW:** Parse Apache, Nginx, and JSON logs with production-ready patterns
4. ⭐ **NEW:** Migrate from ELK/Logstash (Grok to DPL conversion)
5. ⭐ **NEW:** Use the complete Parsing Pattern Library (timestamps, stack traces, HTTP)
6. Implement drop processors for cost optimization
7. Validate parsing success rates and troubleshoot failures

## Processing Stage Overview

The Processing stage is where data transformation happens. It includes three sub-stages executed in order:

```
┌─────────────────────────────────────────────────────────────────┐
│                    PROCESSING STAGE                              │
├─────────────────────────────────────────────────────────────────┤
│                                                                  │
│  1. MASKING (Security First)                                     │
│     └─ Redact sensitive data before any other processing         │
│                                                                  │
│  2. FILTERING (Drop)                                             │
│     └─ Remove unwanted records before further processing         │
│                                                                  │
│  3. PROCESSING (Transform)                                       │
│     ├─ DQL Processors (fieldsAdd, parse, etc.)                  │
```

~~~
│     ├─ Technology Parsers (JSON, Apache, etc.)                   │
│     └─ Custom transformations                                    │
│                                                                  │
└─────────────────────────────────────────────────────────────────┘
```

### Processor Execution Order

Within each sub-stage, processors execute in the order they're defined. You
can reorder them in the UI.

> 💡 **Best Practice:** Order processors logically - parse first, then enrich
with computed fields based on parsed values.

---

## DQL Processor Commands

The DQL processor supports a subset of DQL commands for data transformation.

### Available Commands

| Command | Purpose | Example |
|---------|---------|----------|
| `fieldsAdd` | Add new fields | `fieldsAdd env = "production"` |
| `fieldsRemove` | Remove fields | `fieldsRemove sensitive_field` |
| `fieldsRename` | Rename fields | `fieldsRename old = new_name` |
| `parse` | Extract with DPL | `parse content, "INT:count"` |

### fieldsAdd Examples

#### Static Values
```dql
fieldsAdd environment = "production"
fieldsAdd application = "checkout-service"
fieldsAdd team = "platform-engineering"
```

#### Conditional Values (if/else)
```dql
fieldsAdd severity = if(loglevel == "ERROR", "critical",
                     else: if(loglevel == "WARN", "warning",
                     else: "info"))
```

#### Computed Values
```dql
fieldsAdd message_length = stringLength(content)
~~~

~~~~
fieldsAdd short_host = substring(host.name, 0, 15)
fieldsAdd is_error = loglevel == "ERROR"
```

#### String Operations
```dql
fieldsAdd normalized_status = toLowerCase(status)
fieldsAdd log_prefix = substring(content, 0, 50)
fieldsAdd clean_message = trim(content)
```

#### Coalesce (First Non-Null)
```dql
fieldsAdd effective_level = coalesce(loglevel, status, "UNKNOWN")
```

### fieldsRemove Examples

```dql
// Remove single field
fieldsRemove internal_id

// Remove multiple fields
fieldsRemove temp_field, debug_info, internal_state
```

### fieldsRename Examples

```dql
// Standardize field names
fieldsRename user_id = userId
fieldsRename request_id = requestId
fieldsRename transaction_id = transactionId
```

---

## Dynatrace Pattern Language (DPL)

DPL is a powerful pattern matching language for extracting structured data
from text.

### Core Matchers

| Matcher | Description | Matches |
|---------|-------------|----------|
| `INT` | Integer | `42`, `-17`, `0` |
| `LONG` | Long integer | `1234567890123` |
~~~~

~~~~
| `DOUBLE` | Decimal | `3.14`, `-0.5`, `1.0` |
| `IPADDR` | IP address | `192.168.1.1`, `::1` |
| `IPV4ADDR` | IPv4 only | `10.0.0.1` |
| `IPV6ADDR` | IPv6 only | `2001:db8::1` |
| `LD` | Line data (to delimiter) | Any text until next match |
| `DATA` | Greedy match | Everything remaining |
| `SPACE` | Whitespace | Spaces, tabs |
| `NSPACE` | Non-whitespace | Word-like content |
| `WORD` | Word chars | `hello`, `user123` |
| `JSON` | JSON object | `{"key": "value"}` |
| `EOL` | End of line | Line terminator |

### Pattern Syntax Elements

| Syntax | Meaning | Example |
|--------|---------|----------|
| `MATCHER:field` | Extract to named field | `INT:count` |
| `MATCHER` | Match but don't extract | `SPACE` |
| `MATCHER?` | Optional match | `(':' INT:port)?` |
| `'literal'` | Match exact text | `'error_code='` |
| `(a\|b)` | Alternatives | `('user='\|'userId=')` |
| `MATCHER{n,m}` | Quantifier | `WORD{1,3}` |

### Basic Parse Examples

```dql
// Extract user ID after "user="
parse content, "'user=' LD:user_id"

// Extract error code (integer)
parse content, "'error_code=' INT:error_code"

// Extract IP and port
parse content, "IPADDR:client_ip ':' INT:port"

// Extract JSON payload
parse content, "LD JSON:payload"
```

---

## Real-World Log Format Examples ⭐ NEW

Production-ready DPL patterns for the most common log formats.

### Apache Access Logs (Common Log Format)

**Sample:** `192.168.1.100 - frank [12/Dec/2024:10:30:45 +0000] "GET
~~~~

~~~
/api/users HTTP/1.1" 200 1234`

```dql
parse content, """
  IPADDR:client_ip SPACE '-' SPACE LD:user SPACE
  '[' TIMESTAMP('dd/MMM/yyyy:HH:mm:ss Z'):timestamp ']' SPACE
  '"' LD:method SPACE LD:request_path SPACE LD:protocol '"' SPACE
  INT:status_code SPACE INT:response_bytes
"""
```

**Extracted:** client_ip, user, timestamp, method, request_path, protocol,
status_code, response_bytes

### Nginx (with Response Time)

**Sample:** `192.168.1.50 - - [12/Dec/2024:10:30:45 +0000] "POST
/api/checkout HTTP/1.1" 201 456 "-" "Mozilla/5.0" "0.342"`

```dql
parse content, """
  IPADDR:client_ip SPACE '-' SPACE '-' SPACE
  '[' TIMESTAMP('dd/MMM/yyyy:HH:mm:ss Z'):timestamp ']' SPACE
  '"' LD:method SPACE LD:request_path SPACE LD:protocol '"' SPACE
  INT:status_code SPACE INT:response_bytes SPACE
  '"' LD:referrer '"' SPACE '"' LD:user_agent '"' SPACE
  '"' DOUBLE:response_time_sec '"'
"""
| fieldsAdd response_time_ms = toInt(response_time_sec * 1000)
```

### JSON Application Logs

**Sample:** `{"timestamp":"2024-12-
12T10:30:45.123Z","level":"ERROR","service":"payment-api","message":"Payment
gateway timeout"}`

**Option 1: Use Technology Parser** (Recommended)
- Add **Technology** processor → Select **JSON** parser
- All fields automatically flattened

**Option 2: DQL Parsing**
```dql
parse content, "JSON:log_data"
| fieldsAdd service = log_data["service"]
| fieldsAdd message = log_data["message"]
| fieldsAdd level = log_data["level"]
```
~~~

~~~
### Syslog (RFC 3164)

**Sample:** `<34>Dec 12 10:30:45 webserver sshd[1234]: Failed password for
admin from 192.168.1.100`

```dql
parse content, """
  '<' INT:priority '>'
  TIMESTAMP('MMM dd HH:mm:ss'):timestamp SPACE
  LD:hostname SPACE LD:app_name '[' INT:pid ']:' SPACE
  DATA:message
"""
| fieldsAdd facility = toInt(priority / 8)
| fieldsAdd severity = toInt(priority % 8)
```

### Java Application Logs (Log4j)

**Sample:** `2024-12-12 10:30:45,123 [http-nio-8080-exec-5] ERROR
com.example.Service - Payment failed`

```dql
parse content, """
  TIMESTAMP('yyyy-MM-dd HH:mm:ss,SSS'):log_timestamp SPACE
  '[' LD:thread ']' SPACE
  LD:level SPACE LD:logger SPACE '-' SPACE
  DATA:message
"""
```

**Stack Trace Extraction:**
```dql
parse content, "LD:exception_class ':' SPACE LD:exception_message EOL"
| parse content, "'at ' LD:error_location '(' LD:file ':' INT:line_number
')'"
```

### Kubernetes / Container Logs

**Sample:** `2024-12-12T10:30:45.123456789Z stdout F
{"level":"info","msg":"Request processed"}`

```dql
parse content, """
  TIMESTAMP('yyyy-MM-dd\'T\'HH:mm:ss.SSSSSSSSSXXX'):k8s_timestamp SPACE
  LD:stream SPACE LD:log_tag SPACE
  JSON:log_payload
~~~

~~~
"""
| fieldsAdd level = log_payload["level"]
| fieldsAdd msg = log_payload["msg"]
```

---



---

## Common Parsing Patterns

### Apache/Nginx Access Logs

**Sample log:**
```
192.168.1.100 - - [12/Dec/2024:10:30:45 +0000] "GET /api/users HTTP/1.1" 200
1234
```

**DPL Pattern:**
```dql
parse content, "IPADDR:client_ip SPACE '-' SPACE LD:user SPACE '['
LD:timestamp ']' SPACE '\"' LD:method SPACE LD:path SPACE LD:protocol '\"'
SPACE INT:status_code SPACE INT:bytes"
```

### Key-Value Logs

**Sample log:**
```
userId=12345, action=login, status=success, duration=150ms
```

**DPL Patterns:**
```dql
// Extract each key-value pair
parse content, "'userId=' INT:user_id"
parse content, "'action=' LD:action ','"
parse content, "'status=' LD:status ','"
parse content, "'duration=' INT:duration_ms 'ms'"
```

### Flexible User ID Extraction

**Sample logs with varying formats:**
```
Processing request for user=john123
~~~

~~~
User userId=john123 authenticated
Request from user_id=john123 received
```

**DPL Pattern (alternatives):**
```dql
parse content, "('user='|'userId='|'user_id=') LD:user_id"
```

### Timestamp Parsing

**DPL with timestamp format:**
```dql
parse content, "TIMESTAMP('yyyy-MM-dd HH:mm:ss'):log_timestamp"
parse content, "TIMESTAMP('dd/MMM/yyyy:HH:mm:ss Z'):apache_time"
```

### Optional Fields

**Sample log:**
```
Request to server:8080 completed
Request to server completed
```

**DPL Pattern (optional port):**
```dql
parse content, "'Request to ' LD:server (':' INT:port)? ' completed'"
```

### Stack Trace Extraction

**DPL Pattern:**
```dql
parse content, "LD:exception_class ':' LD:exception_message"
```

---

## Data Transformation Examples

### Example 1: Parse and Enrich Application Logs

**Input:**
```json
{"content": "[2024-12-12T10:30:45] ERROR PaymentService - Payment failed for
orderId=12345, amount=99.99"}
```
~~~

~~~
**Processor 1: Parse log structure**
```dql
parse content, "'[' TIMESTAMP('yyyy-MM-dd\'T\'HH:mm:ss'):log_time ']' SPACE
LD:level SPACE LD:service ' - ' DATA:message"
```

**Processor 2: Extract payment details**
```dql
parse content, "'orderId=' INT:order_id ','"
| parse content, "'amount=' DOUBLE:amount"
```

**Processor 3: Add computed fields**
```dql
fieldsAdd severity = if(level == "ERROR", "critical", else: "normal")
| fieldsAdd application_tier = "payment"
```

### Example 2: JSON Log Processing

**Input:**
```json
{"content": "{\"level\":\"info\",\"msg\":\"Request
processed\",\"duration_ms\":150}"}
```

**Processor: Parse JSON and flatten**
```dql
parse content, "JSON:json_data"
```

Using a Technology Parser (JSON) is often easier for JSON logs.

### Example 3: Multi-Format Log Normalization

**Processor: Normalize different log level formats**
```dql
fieldsAdd normalized_level = if(contains(content, "ERROR") OR
contains(content, "error"), "ERROR",
                             else: if(contains(content, "WARN") OR
contains(content, "warn"), "WARN",
                             else: if(contains(content, "INFO") OR
contains(content, "info"), "INFO",
                             else: if(contains(content, "DEBUG") OR
contains(content, "debug"), "DEBUG",
                             else: "UNKNOWN"))))
```
~~~

~~~
---

## Parsing Pattern Library ⭐ NEW

### Timestamp Patterns (10+ Formats)

| Format | Example | DPL |
|--------|---------|-----|
| ISO 8601 | `2024-12-12T10:30:45Z` | `TIMESTAMP('yyyy-MM-dd\'T\'HH:mm:ssXXX')` |
| ISO + MS | `2024-12-12T10:30:45.123Z` | `TIMESTAMP('yyyy-MM-dd\'T\'HH:mm:ss.SSSXXX')` |
| Apache | `12/Dec/2024:10:30:45 +0000` | `TIMESTAMP('dd/MMM/yyyy:HH:mm:ss Z')` |
| Syslog | `Dec 12 10:30:45` | `TIMESTAMP('MMM dd HH:mm:ss')` |
| Java | `2024-12-12 10:30:45,123` | `TIMESTAMP('yyyy-MM-dd HH:mm:ss,SSS')` |
| MySQL | `2024-12-12 10:30:45` | `TIMESTAMP('yyyy-MM-dd HH:mm:ss')` |
| Unix Epoch | `1702380645` | `LONG:epoch` → `toTimestamp(epoch * 1000)` |

### HTTP Request Patterns

```dql
// Full request line
parse content, "'"' LD:method SPACE LD:path SPACE LD:protocol '"'"

// Separate path and query
parse content, "'"' LD:method SPACE LD:path ('?' LD:query)? SPACE LD:protocol '"'"

// RESTful API paths (/api/v1/users/12345)
parse content, "'/api/v' INT:api_version '/' LD:resource '/' INT:id"
```

### Key-Value Patterns

```dql
// Simple: user=john count=42
parse content, "'user=' LD:user SPACE 'count=' INT:count"

// Quoted: user="John Doe" email="john@example.com"
parse content, "'user="' LD:user '"' SPACE 'email="' LD:email '"'"

// Logfmt: level=info msg="OK" duration=150ms
parse content, "'level=' LD:level SPACE 'msg="' LD:msg '"' SPACE 'duration=' INT:dur 'ms'"
```
~~~

~~~
### Stack Trace Patterns

```dql
// Java exception
parse content, "LD:exception_class ':' SPACE LD:exception_message EOL"

// Stack trace line
parse content, "'at ' LD:class_method '(' LD:file ':' INT:line ')'"

// Caused by
parse content, "'Caused by: ' LD:caused_by ':' SPACE LD:message"

// Python traceback
parse content, "'File "' LD:file '", line ' INT:line ', in ' LD:function"
```

### PII Masking Patterns

```dql
// Credit cards (1234-5678-9012-3456 → ****-****-****-****)
fieldsAdd content = replaceAll(content, "\\b\\d{4}[\\s-]?\\d{4}[\\s-]?\\d{4}[\\s-]?\\d{4}\\b", "****-****-****-****")

// Partial masking (keep last 4)
fieldsAdd content = replaceAll(content, "\\b(\\d{4})[\\s-]?(\\d{4})[\\s-]?(\\d{4})[\\s-]?(\\d{4})\\b", "****-****-****-$4")

// Email addresses
fieldsAdd content = replaceAll(content, "\\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\\.[A-Z|a-z]{2,}\\b", "***@***.***")

// SSN (123-45-6789 → ***-**-****)
fieldsAdd content = replaceAll(content, "\\b\\d{3}-\\d{2}-\\d{4}\\b", "***-**-****")
```

### Network Patterns

```dql
// IP and port
parse content, "IPADDR:ip ':' INT:port"

// IPv4 only
parse content, "IPV4ADDR:ipv4"

// IPv6 only
parse content, "IPV6ADDR:ipv6"
~~~

~~~
// URL parsing
parse content, "LD:protocol '://' LD:hostname (':' INT:port)? LD:path ('?'
LD:query)?"
```

---



---

## ELK/Logstash Migration Patterns ⭐ NEW

### Grok vs. DPL: Key Differences

| Grok (Logstash) | DPL (OpenPipeline) |
|-----------------|-------------------|
| `%{IP:client_ip}` | `IPADDR:client_ip` |
| `%{INT:count}` | `INT:count` |
| `%{WORD:user}` | `WORD:user` |
| `%{GREEDYDATA:msg}` | `DATA:msg` |
| `\\s+` | `SPACE` |

### Apache Log Migration

**Grok:** `%{COMBINEDAPACHELOG}`

**DPL:**
```dql
parse content, """
  IPADDR:client_ip SPACE LD:ident SPACE LD:auth SPACE
  '[' TIMESTAMP('dd/MMM/yyyy:HH:mm:ss Z'):timestamp ']' SPACE
  '"' LD:method SPACE LD:request_path (SPACE LD:http_version)? '"' SPACE
  INT:status_code SPACE (INT:response_bytes | '-') SPACE
  '"' LD:referrer '"' SPACE '"' LD:user_agent '"'
"""
```

### Logstash Filter → OpenPipeline Mapping

| Logstash Filter | OpenPipeline |
|-----------------|-------------|
| `grok { }` | DQL parse processor |
| `mutate { add_field }` | `fieldsAdd field = "value"` |
| `mutate { remove_field }` | `fieldsRemove field` |
| `mutate { rename }` | `fieldsRename old = new` |
| `drop { }` | Drop processor |
| `json { }` | JSON Technology Parser |
| `date { }` | TIMESTAMP in parse |
~~~

~~~~
### Complete Migration Example

**Logstash:**
```ruby
filter {
  if [level] == "DEBUG" { drop { } }
  grok { match => { "message" => "%{TIMESTAMP_ISO8601:ts} %{LOGLEVEL:level} %{GREEDYDATA:msg}" } }
  mutate { add_field => { "env" => "production" } }
}
```

**OpenPipeline:**
1. Drop processor: `loglevel == "DEBUG"`
2. DQL parse: `parse content, "TIMESTAMP('yyyy-MM-dd HH:mm:ss'):ts SPACE LD:level SPACE DATA:msg"`
3. DQL add: `fieldsAdd env = "production"`

---



---

## Technology Bundle Parsers

OpenPipeline includes built-in parsers for common log formats.

### Available Technology Parsers

| Parser | Log Format | Extracted Fields |
|--------|------------|------------------|
| **Apache** | Apache access logs | client_ip, method, path, status, bytes |
| **Nginx** | Nginx access logs | Similar to Apache |
| **JSON** | JSON-formatted logs | All JSON fields flattened |
| **Syslog** | RFC 3164/5424 syslog | facility, severity, hostname, message |
| **Log4j** | Java Log4j format | level, logger, thread, message |
| **AWS CloudWatch** | AWS logs | AWS-specific fields |

### When to Use Technology Parsers

| Use Technology Parser | Use Custom DQL/DPL |
|----------------------|--------------------|
| Standard log format | Custom/proprietary format |
| Quick setup needed | Specific field extraction |
| Common technology | Conditional parsing |
| All fields needed | Selective extraction |
~~~~

~~~~
### Configuring Technology Parsers

1. Open pipeline in OpenPipeline settings
2. Go to **Processing** tab
3. Click **+ Processor** → **Technology**
4. Select parser type
5. Configure matching condition
6. Save

---

## Drop Processors

Drop processors remove records from the pipeline before storage.

### Common Drop Patterns

| Use Case | Matching Condition |
|----------|---------------------|
| Debug logs | `loglevel == "DEBUG"` |
| Trace logs | `loglevel == "TRACE"` |
| Health checks | `contains(content, "health")` |
| Readiness probes | `contains(content, "/ready")` |
| Metrics endpoints | `contains(content, "/metrics")` |
| Heartbeats | `contains(content, "heartbeat")` |
| Specific source | `log.source == "noisy-service"` |

### Drop Processor Configuration

```
Processor Type: Drop
Name: Drop debug logs
Matching Condition: loglevel == "DEBUG" OR status == "DEBUG"
```

### Combining Drop Conditions

```
// Drop all non-essential logs
loglevel == "DEBUG"
  OR loglevel == "TRACE"
  OR contains(content, "health")
  OR contains(content, "/metrics")
```

> ⚠️ **Important:** Dropped data is gone forever. Test drop conditions
carefully before deploying.
~~~~

~~~
---

## Advanced Processing Patterns

### Pattern 1: Fix Missing Timestamp

When logs have a custom timestamp format that's not recognized:

```dql
// Parse timestamp from content
parse content, "'[' TIMESTAMP('yyyy-MM-dd HH:mm:ss'):parsed_timestamp ']'"
```

### Pattern 2: Fix Missing Log Level

When logs don't have a standard loglevel field:

```dql
// Extract level from content
fieldsAdd loglevel = if(contains(content, "[ERROR]") OR contains(content,
"ERROR:"), "ERROR",
                     else: if(contains(content, "[WARN]") OR
contains(content, "WARN:"), "WARN",
                     else: if(contains(content, "[INFO]") OR
contains(content, "INFO:"), "INFO",
                     else: if(contains(content, "[DEBUG]") OR
contains(content, "DEBUG:"), "DEBUG",
                     else: "NONE"))))
```

### Pattern 3: Parse JSON from Within Text

When JSON is embedded in a log message:

```dql
// Extract JSON from text
parse content, "LD JSON:embedded_json LD"
```

### Pattern 4: Compute Response Time Categories

```dql
fieldsAdd response_category = if(duration_ms < 100, "fast",
                              else: if(duration_ms < 500, "normal",
                              else: if(duration_ms < 1000, "slow",
                              else: "very_slow")))
```
~~~

~~~
### Pattern 5: Standardize Boolean Fields

```dql
fieldsAdd is_success = if(status == "success" OR status == "ok" OR status ==
"200", true, else: false)
```

### Pattern 6: Extract Service Name from Path

```dql
// From /api/v1/users → users
parse content, "'/api/v' INT '/' LD:service_name"
```

---

## Validating Your Processing

After configuring processors, validate that parsing is working correctly.

```python
// Check parsing success rate across all pipelines
fetch logs, from: now() - 1h
| summarize {
    total = count(),
    with_loglevel = countIf(isNotNull(loglevel)),
    with_status = countIf(isNotNull(status)),
    with_either = countIf(isNotNull(loglevel) OR isNotNull(status))
  }
| fieldsAdd parsing_rate = round((toDouble(with_either) / toDouble(total)) *
100, decimals: 1)
```

```python
// Parsing success rate by pipeline
fetch logs, from: now() - 1h
| filter isNotNull(dt.openpipeline.pipelines)
| summarize {
    total = count(),
    parsed = countIf(isNotNull(loglevel))
  }, by: {dt.openpipeline.pipelines}
| fieldsAdd parsing_rate = round((toDouble(parsed) / toDouble(total)) * 100,
decimals: 1)
| sort parsing_rate asc
```

```python
// Sample logs that failed parsing (no loglevel extracted)
~~~

~~~
fetch logs, from: now() - 1h
| filter isNull(loglevel) AND isNull(status)
| fields timestamp, log.source, dt.openpipeline.pipelines, content
| limit 25
```

```python
// Verify specific parsed fields exist
// Replace 'your_field' with fields your parsing should create
fetch logs, from: now() - 1h
| filter isNotNull(dt.openpipeline.pipelines)
| summarize {
    total = count(),
    with_user_id = countIf(isNotNull(user_id)),
    with_request_id = countIf(isNotNull(request_id))
  }, by: {dt.openpipeline.pipelines}
```

```python
// Sample successfully parsed logs to verify field extraction
fetch logs, from: now() - 1h
| filter isNotNull(loglevel)
| limit 20
```

```python
// Verify drop processors are working
// If drops are configured, debug log count should be low
fetch logs, from: now() - 1h
| summarize {debug_count = countIf(loglevel == "DEBUG")}
| fieldsAdd message = if(debug_count == 0, "✅ Drop processor working - no
debug logs",
                       else: "⚠️ Debug logs still present - check drop
configuration")
```

```python
// Check log level distribution after processing
fetch logs, from: now() - 1h
| summarize {log_count = count()}, by: {loglevel}
| sort log_count desc
```

---

## DPL Architect Tool

Dynatrace provides a **DPL Architect** tool for building and testing
~~~

~~~
patterns:

### Accessing DPL Architect

1. Navigate to **Settings → OpenPipeline**
2. When adding a parse processor, click **Open DPL Architect**
3. Or access via: `https://{your-
environment}.apps.dynatrace.com/ui/apps/dynatrace.dpl.architect`

### Using DPL Architect

1. Paste sample log content
2. Build pattern interactively
3. See extracted fields in real-time
4. Copy pattern to processor definition

> 💡 **Tip:** Always test patterns in DPL Architect before deploying to
production pipelines.

---

## Complete Processing Pipeline Example

### Pipeline: `application-logs`

**Processor 1: Drop Debug (Drop)**
```
Matching: loglevel == "DEBUG" OR contains(content, "[DEBUG]")
```

**Processor 2: Parse Application Log (DQL)**
```dql
parse content, "'[' TIMESTAMP('yyyy-MM-dd HH:mm:ss'):log_ts ']' SPACE '['
LD:level ']' SPACE '[' LD:thread ']' SPACE LD:class ' - ' DATA:message"
```

**Processor 3: Extract Request ID (DQL)**
```dql
parse content, "'requestId=' LD:request_id"
```

**Processor 4: Add Environment Tags (DQL)**
```dql
fieldsAdd environment = "production"
| fieldsAdd application = "checkout-service"
| fieldsAdd team = "platform"
```
~~~

~~~
**Processor 5: Compute Severity (DQL)**
```dql
fieldsAdd severity = if(level == "ERROR", "P1",
                     else: if(level == "WARN", "P2",
                     else: "P3"))
```

---

## Next Steps

Now that you can transform data, continue with:

| Notebook | Focus Area |
|----------|------------|
| **OPMIG-07** | Metric & Event Extraction |
| **OPMIG-08** | Security, Masking & Compliance |
| **OPMIG-09** | Troubleshooting & Validation |

---

## References

- [OpenPipeline Processing](https://docs.dynatrace.com/docs/discover-
dynatrace/platform/openpipeline/concepts/processing)
- [Processing Examples](https://docs.dynatrace.com/docs/discover-
dynatrace/platform/openpipeline/use-cases/processing-examples)
- [Dynatrace Pattern Language](https://docs.dynatrace.com/docs/discover-
dynatrace/platform/grail/dynatrace-pattern-language)
- [DPL Architect Tool](https://docs.dynatrace.com/docs/discover-
dynatrace/platform/grail/dynatrace-pattern-language/dpl-architect)
- [DQL Functions in OpenPipeline](https://docs.dynatrace.com/docs/discover-
dynatrace/platform/openpipeline/reference/openpipeline-dql-functions)

---

*Last Updated: December 12, 2025*
~~~