Groovy in SAP CPI — Handling Large Payloads Efficiently

Large payloads (XML, CSV, JSON, PDF, ZIP, Base64) can cause:

- Out-of-memory errors
- Slow execution
- Timeouts
- Unnecessary CPU load

Groovy in CPI is powerful, but must be used carefully.

Below are the most important techniques to keep performance high.

1. DO NOT load the entire payload into memory unnecessarily

Avoid:

```
def body = message.getBody(String)      // May explode memory for 100MB+
```

Prefer working with **streams**:

```
def is = message.getBody(java.io.InputStream)
```

This is far more memory-efficient.

2. Streaming XML Processing (Large Files)

Using `XmlSlurper` on huge XML is dangerous because it builds a full DOM tree in memory.

Instead, use **XmlParser with SAX/streaming mode**:

✔ Efficient XML Iteration (SAX)

```
import javax.xml.stream.XMLInputFactory
import javax.xml.stream.XMLStreamConstants

def Message processData(Message message) {
    def inputStream = message.getBody(InputStream)
    def reader =
XMLInputFactory.newInstance().createXMLStreamReader(inputStream)

    while(reader.hasNext()) {
        if (reader.next() == XMLStreamConstants.START_ELEMENT &&
            reader.getLocalName() == "Record") {

            // Process each <Record> element without loading whole XML
            println "Found a record"
        }
    }

    return message
}
```

When to use this?

- XML > 5MB
- Repeated element structures
- Mapping or filtering tasks

---

3. Streaming CSV Processing

Never convert full CSV to string when large.

✔ Stream line-by-line (memory safe)

```
import java.io.BufferedReader
import java.io.InputStreamReader

def Message processData(Message message) {

    def is = message.getBody(InputStream)
    def reader = new BufferedReader(new InputStreamReader(is, "UTF-8"))

    String line
    while ((line = reader.readLine()) != null) {
        // Process each line individually
        println "Line: $line"
    }

    return message
}
```

Perfect for:

- SFTP files
- DataStore binaries
- JDBC extraction files

---

4. Streaming JSON Processing

SAP CPI does not have Jackson streaming by default, but we can still keep memory optimized:

✔ Don’t parse huge JSON fully with JsonSlurper

❌ Bad for large files:

```
def json = new JsonSlurper().parseText(body)   // loads entire file into
RAM
```

✔ Instead: convert JSON incrementally (chunk method)

When using large arrays:

```
[
  { record… },
  { record… },
  …
  { record… }
]
```

Use a streaming iterator over raw text:

```
import java.util.regex.*

def Message processData(Message message) {
    def is = message.getBody(InputStream)
    def text = is.getText("UTF-8")

    def matcher = Pattern.compile("\\{(.*?)\\}",
Pattern.DOTALL).matcher(text)

    while(matcher.find()) {
        def objText = "{" + matcher.group(1) + "}"
        println("JSON object: $objText")
    }

    return message
}
```

Works for large JSON arrays.

5. Large Base64 Payload Handling

Never do:

```
def decoded = message.getBody(String).decodeBase64()
```

This may allocate a huge byte array.

✔ Stream-safe decoding:

```
import java.util.Base64

def is = message.getBody(InputStream)
def decoder = Base64.getMimeDecoder()
def decodedStream = decoder.wrap(is)

message.setBody(decodedStream)
```

---

6. Writing Output Stream Efficiently

When generating large XML/CSV files:

✔ Do NOT build large strings

❌ Avoid:

```
def output = ""
records.each { output += it + "\n" }
```

✔ Use a streaming writer

```
import java.io.ByteArrayOutputStream
import java.io.OutputStreamWriter

def baos = new ByteArrayOutputStream()
def writer = new OutputStreamWriter(baos, "UTF-8")

records.each { row ->
    writer.write(row + "\n")
}
writer.flush()

message.setBody(baos.toByteArray())
```

---

7. Pagination + Chunking (Best Practice for CPI)

CPI recommends processing large data in **chunks** to prevent memory issues.

✔ Example: Split payload into blocks of 1000 lines

```
def block = []
int chunkSize = 1000
int counter = 0

reader.eachLine { line ->
    block << line
    counter++

    if (counter == chunkSize) {
        // Process the block of 1000 lines
        println "Processing block..."
        block.clear()
        counter = 0
    }
}
```

---

8. Avoid XmlUtil.serialize() on large XML

This converts the entire XML tree to string — not safe for large payloads.

Alternative: write only parts using StreamingMarkupBuilder:

```
def xml = new groovy.xml.StreamingMarkupBuilder().bind {
    Root {
        records.each { rec ->
            Record {
                ID(rec.id)
            }
        }
    }
}
```

---

9. Using GZIP inside CPI for large payload compression

✔ Compress output

```
import java.util.zip.GZIPOutputStream
import java.io.ByteArrayOutputStream

def baos = new ByteArrayOutputStream()
def gzip = new GZIPOutputStream(baos)
gzip.write(message.getBody(byte[]))
gzip.close()

message.setBody(baos.toByteArray())
```

✔ Decompress input

```
import java.util.zip.GZIPInputStream

def gzipStream = new GZIPInputStream(message.getBody(InputStream))
message.setBody(gzipStream)
```

This reduces memory footprint dramatically.

---

10. Avoid Groovy closures inside heavy loops

Closures create overhead.

Prefer **classic loops**:

```
for (int i = 0; i < list.size(); i++) {
    def rec = list[i]
}
```

---

11. Use Maps and Lists Sparingly

Avoid nested maps for very large structures.

Better: stream records one by one → convert → write out.

---

12. Offload Large Logic to Integration Advisor or XSLT

If payload > 10MB:

- Prefer XSLT for XML transformations
- Prefer Integration Advisor for message mapping
  They handle streaming more efficiently than Groovy.

Summary — Best Techniques for Large Payload Handling in CPI

Technique	Benefit
Use InputStream instead of String	Avoids memory explosion
XML streaming (StAX)	No DOM, low memory
CSV line-by-line processing	Lightweight processing
Regex-based JSON streaming	Bypass full JSON parsing
Incremental writing	Avoid huge concatenated strings
Base64 stream decoding	Prevents huge byte-array allocation
Chunking / pagination	Safe for very large files
GZIP compression	Faster movement of huge payloads
