ChatGPT

Efficient Serialization of the “Catchup” Output for LLM Context

Understanding the Current “Catchup” Output Format

The `/catchup` command in Claude Code (and similar AI-IDE setups) is designed to load all recently changed files from your Git branch into the AI’s context [1]. In practice, the **catchup output** is currently a large JSON object (included as *structured content* in the MCP response) that enumerates each changed file and its contents or diff. For example, the output likely looks roughly like:

```
{
  "changedFiles": [
    {
      "path": "src/foo.py",
      "content": "<full text or diff of foo.py>"
    },
    {
      "path": "src/bar.js",
      "content": "<text of bar.js>"
    },
    // ...more files...
  ]
}
```

Each file’s content is embedded as a string in the JSON. This means the model is given the raw text of every modified file, packaged into a single JSON structure. While this ensures Claude has all the details of the changes, it can produce a **huge payload** – on the order of 10k tokens or more for non-trivial changes – which quickly consumes the context window.

Why the Current JSON Format is Token-Heavy

There are two main reasons the catchup output is so large:

- **Volume of Data:** Including the *full content or diff* of each changed file means potentially thousands of lines of code/text are being inlined. Even a few moderately sized files can add up to thousands of tokens. If all that text is inserted at once, the token count will balloon. Ideally, we want to keep the catchup under ~1.5k tokens, as you noted.

- **JSON Formatting Overhead:** The use of JSON adds significant token overhead due to its syntax (braces, quotes, colons, commas) and escaping of characters. If the JSON is pretty-printed or

contains lots of structural characters, the model's tokenizer will count those as extra tokens without adding semantic value [2]. Research from Microsoft shows that *human-friendly JSON formatting (with indentation, newline, and whitespace) wastes a lot of tokens* because each bracket, quote, and whitespace can become a separate token [2]. In their analysis, many tokens in a naive JSON output were just **formatting** – e.g. quotes, commas, and newline/indent characters [2]. These don't help the model understand the content, they just make the JSON readable to humans. For example, the string delimiters and `\n` escape sequences for every line of file content contribute heavily to token count.

- **Duplicate Representation:** By MCP convention, when a tool returns structured data, it often includes it twice: once as machine-readable JSON (in the `structuredContent` field) and **again as a text blob** for backward compatibility [3] [4]. This means the JSON may be present both as a parsed object *and* as a raw JSON string in a `"text"` field of the result. Such duplication literally doubles the tokens used for the same data. If the client is indeed including both, it's a huge inefficiency.

Overall, a large pretty-printed JSON with full file texts will be extremely token-inefficient. In one comparison, a neatly formatted JSON output required ~370 tokens, whereas a more compact format could cut that dramatically (as discussed below) [5] [6].

Strategies to Reduce Token Usage in Catchup

To make the catchup output **much more token-efficient** (targeting ~1.5k tokens max), we can apply several strategies:

1. Minimize or Change the Serialization Format:
Using a more compact serialization can save a lot of tokens:

- **Remove Pretty-Printing:** Ensure the JSON is **minified** – no unnecessary spaces or line breaks. For example, keys and values should follow one after another with no indentation. Eliminating whitespace and newline tokens can significantly shrink size [2]. You might also use shorter key names (e.g. `"f"` instead of `"filename"`, `"c"` for `"content"`) to shave off a few tokens per entry.

- **Compound Tokens:** Some model tokenizers treat common JSON patterns as single tokens (for instance, the sequence `": "` or `"}, "` might tokenize as one unit). By **removing spaces** around colons and commas and not quoting numeric indices, you encourage the model to use such compound tokens [6]. Essentially, producing JSON like `{"path":"file","content":"..."}` `{"path":...}` (all one line or compact) can reduce the count of structural tokens [6].

- **Use YAML or Another Concise Format:** YAML can represent the same data with fewer syntactic characters. In tests, a YAML output used about **30% fewer tokens** than equivalent JSON [5]. For example, YAML can embed multi-line text more naturally (using `|` for block literals) without needing `\n` escapes or quotes around every line of code. A YAML catchup might look like:

```
changedFiles:
  - path: src/foo.py
    content: |
      <file text here, line by line>
  - path: src/bar.js
    content: |
      <file text here>
```

This avoids the JSON string escaping overhead for newlines and quotes. The LLM will parse it just as easily, and you lose many quote and backslash tokens. *Empirical data*: a structured YAML response saved ~100 tokens compared to JSON in one case [5].

- **Custom Delimited Format:** You could go even simpler and use a plain text convention instead of JSON. For instance, list files one after the other with clear delimiters. Example:

```
=== File: src/foo.py ===
<file content here...>
=== File: src/bar.js ===
<file content...>
```

This format has virtually zero overhead per file – just a delimiter line. No braces, no quotes at all. The model should still understand that each section is a file name followed by content. You'd need to ensure the prompt (system or user message) explains this format so Claude knows how to interpret it. This is a trade-off: it loses strict machine structure, but massively compresses token count by dropping all JSON syntax.

2. Limit the Content Included:
Reconsider whether you need the **full content of every changed file** in the context. Often, a diff or a summary of changes might suffice:

- **Use Diffs Instead of Full Files:** If the changes are small relative to file size, feeding just the `git diff` output can be much more concise than the entire file text. Diffs include only added/removed lines (plus some context), which could be far shorter. This way, the AI sees what changed without all the unchanged code. The catchup JSON could then include a `"diff"` field rather than full `"content"`. If a file is entirely new, the diff is basically the file; if minor edits, the diff is tiny.

- **Filter or Prioritize:** If dozens of files changed, perhaps not all are relevant to the current task. You might limit to "important" files (e.g., code files vs. minor config/docs) or recent edits. Shrivu's approach was to **prioritize the most important changed files** for analysis [7]. You could incorporate logic to only include top N files by size or relevance, and just list the names of the rest (the model can always fetch them later if needed).

- **Summarize Large Chunks:** For extremely large diffs or files, a *brief summary* of changes could replace the raw content. For instance, "In `data.csv`, 50 lines were updated (values in column X changed)." However, use caution: summarizing code changes can omit details the model might need

for coding assistance. Only summarize if the exact content isn't critical to the model's task, or if you plan to retrieve details on demand later.

3. Avoid Duplicate Representations:
Ensure the catchup data is only included **once**. If your MCP server or workflow currently outputs the JSON both as `structuredContent` and as a raw text string, try to eliminate the redundant copy. According to the MCP spec, the structured JSON is often mirrored in a text block for backward compatibility [3]. In a modern client like Claude Code, you can probably rely on the structured form alone. Configuring the client or server to drop the `"text": "{...json...}"` field could nearly halve the token count. Double-check how the Emacs MCP server is returning data: if you see a `content: [{type: "text", text: "...json..."}]` alongside `structuredContent`, that's the duplication to remove [4] [8].

4. Use External References (Lazy Loading):
Rather than stuffing all file content into the context at once, consider using **resource links or IDs** that Claude can fetch if needed. The Model Context Protocol supports returning a `resource_link` – essentially a URI pointer to a file [9]. For example, the catchup result could list changed file paths with `type: "resource_link"` and a file URI, instead of inlining content. Claude (the client) would see the list of filenames and know it can call a `resources/read` tool to load any of those files on demand. This way, the initial context stays light (just file names, which are short), and only if the model's next steps require a file's content will it retrieve it. This **lazy loading** approach can keep the main prompt under 1.5k tokens easily. The downside is it requires the agent to proactively fetch content when needed (or you write the workflow to automatically fetch the most critical ones). If your workflow can orchestrate it, this is very powerful: e.g., "List 10 changed files; then immediately fetch the 2 most important ones" – you'd pay tokens for only those two files rather than all ten.

5. Beware of Binary/Encoded Formats:
You asked if an LLM can store/retrieve a "binary-kind of format" in its memory. Essentially, feeding compressed or binary data (like a gzipped blob or base64 string) to an LLM is not useful because the model can't inherently decode it**. The model only "sees" a sequence of tokens – it has no built-in ZIP or base64 decoder tool (unless you explicitly implement one within the conversation). In fact, binary data often ends up as high-entropy tokens that the model doesn't recognize as meaningful patterns, and base64-encoding can *increase* token count by ~33% while still being opaque to the model. So, compressing the data outside the model and sending it in binary form will just produce a shorter-looking token sequence that the LLM can't interpret accurately. The only way an LLM could decompress something is if it was taught the algorithm step-by-step in the prompt (which would itself cost many tokens and be brittle). Therefore, focus on** human-readable but compact encodings**, as discussed above, rather than true binary storage. In short,** no, current LLMs don't have a "binary memory" you can leverage in context – *their context is* tokenized text.

Putting It All Together

By restructuring the catchup output with the above techniques, you can drastically reduce its token footprint:

- Switch to a slim format (minified JSON, YAML, or even custom delimiters) to cut JSON syntactic bloat. Studies show moving away from verbose JSON can easily save 30–40% tokens [5] [6].

- Only include essential content (diffs or selected files) instead of everything, wherever possible.
- Eliminate any duplicated data in the tool response.
- Take advantage of MCP's ability to reference external resources [9], so not all data travels with every prompt.

As a result, your **catchup** output could shrink from ~10.6k tokens to a fraction of that. For example, a concise list of file diffs in YAML might come out to well under 1.5k tokens, achieving the efficiency target. Not only does this avoid blowing the context window, it also **speeds up processing** (fewer tokens means faster responses and lower cost, if using API calls).

In summary, the catchup output currently contains JSON-formatted file data which is comprehensive but very verbose. To meet the 1.5k token goal, you'll want to **serialize that data more efficiently** and judiciously trim what's included. By doing so, you maintain useful context for the model without drowning it (and your context budget) in unnecessary bulk.

Sources:

- Anthropic Claude Code documentation – description of `/catchup` functionality [1].
- Bryce Williams et al., *"Token efficiency with structured output from language models"* – analysis of JSON vs YAML token usage and JSON formatting overhead [2] [6] [5].
- Model Context Protocol Spec – guidelines on structured content and resource linking in tool outputs [3] [9].
- Shrivu Shankar, *"How I Use Every Claude Code Feature"* – notes on using `/clear` + `/catchup` and focusing on important changed files [7].

---

[1] Claude Code Cheatsheet, A Complete Beginners Guide for Developers
https://apidog.com/blog/claude-code-cheatsheet/

[2] [5] [6] Token efficiency with structured output from language models | by Bwilliams | Data Science + AI at Microsoft | Medium
https://medium.com/data-science-at-microsoft/token-efficiency-with-structured-output-from-language-models-be2e51d3d9d5

[3] [4] [8] [9] Tools - Model Context Protocol
https://modelcontextprotocol.io/specification/draft/server/tools

[7] How I Use Every Claude Code Feature - Shrivu's Substack
https://blog.sshh.io/p/how-i-use-every-claude-code-feature/comments