
GCF: A Token-Optimized Wire Format for Structured LLM Interactions

Dayna Blackwell, Blackwell Systems

2026-06-06 · DOI: [10.5281/zenodo.20579817](https://doi.org/10.5281/zenodo.20579817)

Contents

GCF: A Token-Optimized Wire Format for Structured LLM Interactions	2
Abstract	2
1. Why JSON Fails at Scale	3
1.1 Why This Matters Now	3
1.2 Why Not Just Use Binary?	4
2. Design Principles	4
2.1 Referential Identity	4
2.2 Topological Encoding	5
2.3 Hierarchical Grouping	5
3. Specification	5
3.1 Grammar	5
3.2 Header	6
3.3 Node Lines	6
3.4 Edge Lines	7
3.5 Group Headers	7
3.6 Generic Profile (Generic Encoding)	8
3.7 Session Statefulness	9
3.8 Streaming Encoding Extension	9
4. Implementation Status	10
Correctness Validation	11
Production Deployment	12
5. Where Token Savings Come From	12
5.1 Generic Profile Savings	12
6. Benchmarks	13
6.1 Token Comparison	13
6.1a Comprehension Accuracy at Scale (500 symbols, multi-model)	14
6.2 Session Statefulness Savings	18
7. Comparison to Alternatives	19
7.1 Columnar/TSV	19
7.2 Binary Formats (Protobuf, MessagePack, FlatBuffers)	19
7.3 JSON-LD / RDF	19
7.4 Markdown / Freeform Text	19
7.5 TOON (Token-Oriented Object Notation)	19
7.6 Custom Compressed JSON	21
8. Limitations and Validation	21
9. Implications for MCP and Agent Tooling	22
9.1 LLM Generation: GCF as a Bidirectional Format	22
9.2 Streaming: Progressive Context Delivery	25
9.3 Delta Encoding	26
10. Conclusion	26
Reference Implementation	27

Appendix A: Full Example	28
Appendix B: Streaming Example	29

GCF: A Token-Optimized Wire Format for Structured LLM Interactions

Dayna Blackwell, Blackwell Systems · dayna@blackwell-systems.com

Date: 2026-06-11 (v3) · **DOI:** [10.5281/zenodo.20579817](https://doi.org/10.5281/zenodo.20579817)

Abstract

AI agents consume and produce structured data under fixed token budgets. The dominant encoding is JSON, which wastes 75%+ of tokens on structural overhead: field names, delimiters, and repeated identifiers. We present GCF, a bidirectional text-based wire format for LLM interactions. GCF supports two encoding profiles: a **graph profile** exploiting referential identity (local IDs), topological encoding (edge arrows), and hierarchical grouping (section headers); and a **generic profile** encoding arbitrary structured data with positional rows, pipe separators, and inline primitive arrays.

We evaluated GCF across 1,300+ LLM evaluations spanning 10 models and 3 providers (Anthropic, OpenAI, Google). No model has been trained on GCF.

We benchmark against JSON and TOON (Token-Oriented Object Notation), a tabular encoding format that declares array field names once and uses comma-separated rows, achieving 30-60% savings versus JSON. TOON is the closest competitor to GCF in the LLM wire format space.

Comprehension: 23 runs across 10 models. GCF averages 90.7% accuracy where TOON averages 68.5% and JSON averages 53.6%. Four models achieve 100% (Claude Sonnet 4.6, Gemini 2.5 Pro, Gemini 3.1 Pro, Gemini 3.5 Flash). GCF wins 22 of 23 runs (1 tie, 0 losses).

Generation: 28 runs across 9 models. GCF achieves 5/5 validity on every frontier model. TOON's official decoder rejects LLM-generated output on 7 of 9 models due to a structural design flaw in flat tabular encoding. GCF output is 63% smaller than JSON and 33% smaller than TOON.

Token efficiency: GCF achieves 79% fewer input tokens than JSON at 500 symbols. On TOON's own benchmark (their datasets, their tokenizer, their methodology), GCF wins all 6 datasets.

Session deduplication (92.7% savings by the 5th call) and delta encoding (81.2% on re-queries) compound savings across multi-turn interactions. A streaming encoding extension enables zero-buffering encode with $O(1)$ memory per row. The format is implemented in six languages (all at v1.0.0+), verified across 200M+ property-based round-trips and 7.9M fuzz executions, with a cross-language 5x5 encode/decode matrix passing. A bidirectional MCP proxy with session dedup, HTTP backend/frontend, and response caching enables zero-code

adoption. JSON Schema validation works on decoded output unchanged. Specification v2.0 Stable: gcformat.com.

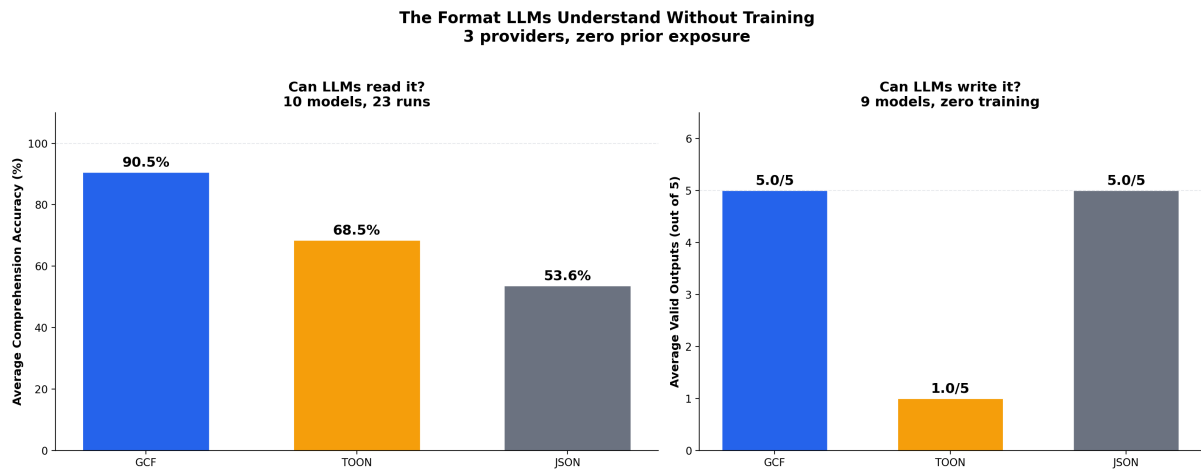


Figure 1: Comprehension and Generation across 10 models and 3 providers

1. Why JSON Fails at Scale

The Model Context Protocol (MCP) defines how AI agents interact with external tools. Tool responses are overwhelmingly encoded as JSON. This is convenient for developers but expensive for the consumer that matters most: the language model itself.

Consider a typical MCP tool response returning 10 graph nodes and 8 edges (a blast radius query, a dependency subgraph, or a context retrieval result). In JSON, this payload consumes ~965 tokens. The same semantic content in GCF consumes ~233 tokens. The difference, 732 tokens, is pure waste: field name repetition, structural delimiters (`{`, `}`, `[`, `]`, `:`, `"`, `"`), and full qualified names repeated in every edge reference.

This waste compounds across a task. An agent making 5 tool calls during a code change task consumes ~4,825 tokens on JSON tool responses. In GCF, the same 5 calls consume ~1,165 tokens, and less with session statefulness enabled (previously-transmitted nodes are referenced by local ID without retransmission). The difference, ~3,660 tokens, is context window capacity that could hold source code, documentation, or additional tool results.

1.1 Why This Matters Now

Three trends make this problem urgent:

1. **Tool-heavy agent workflows.** Modern AI agents make 10-50 tool calls per task. Each call returns JSON. Token budgets are consumed by tool overhead before the agent finishes its work.

2. **Graph-structured responses are growing.** Code intelligence, dependency analysis, knowledge graphs, and system topology queries all return graph data. Graph payloads are JSON's worst case because they contain repeated node references across edges.
3. **Context window costs are real.** Whether measured in dollars (API billing), latency (time-to-first-token), or capability (what fits in the window), every wasted token reduces the agent's effectiveness.

1.2 Why Not Just Use Binary?

Binary formats (Protocol Buffers, MessagePack, FlatBuffers) optimize for byte size and decode speed. But LLMs consume text, not bytes. A protobuf-encoded tool response must be decoded to text before the LLM can process it, and the decoded text is typically JSON, eliminating the savings at the point of consumption.

The optimization target is not bytes on wire. It is tokens in the context window. These are different quantities with different solutions.

2. Design Principles

GCF's graph profile is designed around three observations about graph data that JSON cannot exploit. Its generic profile (Section 3.6) generalizes these principles to arbitrary structured data.

2.1 Referential Identity

Graph nodes are referenced multiple times: once in their declaration and once per edge they participate in. In JSON, each reference repeats the full qualified name:

```
{
  "source": "github.com/org/repo/internal/mcp.NewServer",
  "target": "github.com/org/repo/internal/mcp.requireHash",
  "edge_type": "calls"
}
```

In GCF, nodes are declared once with a local ID and referenced by that ID thereafter:

```
@0 fn github.com/org/repo/internal/mcp.requireHash 0.78 lsp_resolved
@4 fn github.com/org/repo/internal/mcp.NewServer 0.54 lsp_resolved
@0<@4 calls
```

The full qualified name appears once per node. Every subsequent reference is 2-3 tokens (@0, @4) instead of 15-20 tokens.

2.2 Topological Encoding

JSON encodes edges as objects with named fields. GCF encodes edges as directed connections:

JSON (1 edge, ~12 tokens):

```
{"source": "...", "target": "...", "edge_type": "calls"}
```

GCF (1 edge, ~5 tokens):

```
@0<@4 calls
```

The < arrow encodes direction. The source and target are local IDs. The edge type is a bare token. No field names, no delimiters, no quoting.

2.3 Hierarchical Grouping

Graph query results often partition nodes by distance from a query center: direct targets (distance 0), related symbols (distance 1), extended context (distance 2+). In JSON, each node carries a "distance": N field. In GCF, a section header replaces all per-node distance fields:

```
## targets
@0 fn ... 0.78 lsp_resolved
@1 method ... 0.74 lsp_resolved
## related
@4 fn ... 0.54 lsp_resolved
```

One header replaces N repeated fields.

3. Specification

3.1 Grammar

```
payload      = header LF { section } [ summary ] ;
section      = group-header LF { line LF } ;
line         = node-line | edge-line | ref-line | tabular-row
              | kv-line | nested-ref | inline-array | comment ;
summary      = "##! summary" SP key-value { SP key-value } LF ;

header       = "GCF" SP key-value { SP key-value } ;
```

```

group-header  = "##" SP group-name [ SP "[" count-or-deferred "]" [ field-
decl ] ] ;
count-or-deferred = count | "?" ;
field-decl     = "{" field-name { "," field-name } "}" ;
node-line      = "@" id SP kind SP qname SP score SP provenance ;
edge-line      = "@" target "<" "@" source SP edge-type [ SP status ] ;
ref-line       = "@" id SP SP "# previously transmitted" ;
tabular-row    = [ "@" id SP ] value { "|" value } ;
kv-line        = key "=" value ;
inline-array   = key "[" count-or-deferred "]" ":" SP value { "," value } ;
nested-ref     = "." field-name ;
comment        = "#" SP text ;

id             = DIGIT { DIGIT } ;
count          = DIGIT { DIGIT } ;
kind           = "fn" | "type" | "method" | "iface" | "var" | "const"
               | "resource" | "table" | "class" | "selector" | "field"
               | "route" | "ext" | "file" | "pkg" | "svc" ;
status        = "added" | "removed" ;

```

3.2 Header

The header line identifies the format and carries payload metadata:

GCF profile=graph tool=context_for_task budget=5000 tokens=1847 symbols=10 edges=8

tool identifies the MCP tool that produced this response. budget and tokens enable the consumer to assess utilization. symbols and edges give counts without scanning.

3.3 Node Lines

```
@{id} {kind} {qualified_name} {score} {provenance}
```

Fields are positional. No field names, no delimiters beyond whitespace. Kind abbreviations reduce token count (fn vs "function", iface vs "interface").

Abbreviation	Full form
fn	function
type	type
method	method

Abbreviation	Full form
iface	interface
var	var
const	const
resource	resource
table	table
class	class
selector	selector
field	field
route	route_handler
ext	external
file	file
pkg	package
svc	service

3.4 Edge Lines

@{target}<@{source} {edge_type} [{status}]

The < arrow points toward the target. @0<@4 calls means “@4 calls @0.” Status is optional: added or removed for diff payloads.

3.5 Group Headers

targets
related
extended
edges [N]

Group headers partition the payload into semantic sections. The group a node appears in encodes its distance from the query center, eliminating per-node distance fields. The edges section header includes [N] (the edge count) to enable direct count verification by the LLM without scanning.

3.6 Generic Profile (Generic Encoding)

The graph profile (Sections 3.1-3.5) encodes typed nodes and edges. The generic profile encodes arbitrary structured data using the same grammar primitives.

Tabular arrays:

```
## {name} [{count}][{field1},{field2},{field3}]
value1|value2|value3
```

The header declares field names once. Rows are pipe-separated positional values. No field names repeated per record.

```
## employees [3]{id,name,department,salary}
1|Alice Smith|Engineering|95000
2|Bob Jones|Sales|72000
3|Carol Wu|Marketing|85000
```

Primitive arrays (all elements are scalars) are inlined on a single line:

```
tags[3]: production,us-east-1,critical
ports[3]: 8080,8443,9090
```

Key-value pairs for primitive object fields:

```
config=production
port=5432
active=true
```

Section headers for nested objects:

```
## database
  host=db.example.com
  port=5432
```

Nested fields in tabular rows use @{id} prefixes and .field {}:

```
## orders [2]{id,total,status,customer}
@0 1001|249.99|shipped|^
  .customer {}
    name=Alice Smith
    tier=premium
```

Value encoding rules:

Type	Encoding
String	bare text
Number	unquoted decimal
Boolean	lowercase true/false
Null	-
Empty string	""
String containing or \n	quoted, with \" and \\ escaping

Uniformity detection: An array is tabular-eligible if the first 5 elements are objects with at least 70% key overlap with the first element, accommodating semi-uniform data.

3.7 Session Statefulness

Across multiple tool calls in a session, previously-transmitted nodes can be referenced without retransmission:

```
GCF profile=graph tool=context_for_files tokens=800 symbols=5 edges=1 session=true
## targets
@0 # previously transmitted
@7 fn github.com/org/repo/internal/mcp.handleBlastRadius 0.62 lsp_resolved
## edges [1]
@0<@7 calls
```

The `session=true` header flag enables ID persistence. A bare `@0` (no kind, name, or score) references a node transmitted in a previous response. Multi-call workflows get progressively cheaper as the session builds a shared vocabulary.

This exploits a property unique to agent tool interactions: the consumer (the LLM) maintains conversational state across calls. A traditional wire format cannot assume its consumer remembers previous messages. GCF can because LLM context windows are, by definition, stateful.

3.8 Streaming Encoding Extension

When the encoder does not know payload size upfront (data arriving incrementally from a database cursor, API pagination, or graph traversal), it uses streaming mode:

```
GCF profile=graph tool=context_for_task budget=5000
## targets
@0 fn pkg.Auth 0.95 lsp_resolved
@1 fn pkg.Handler 0.88 lsp_resolved
## related
@2 type pkg.Config 0.72 ast_inferred
## edges [?]
@0<@1 calls
@2<@0 references
##! summary counts=2
```

The [?] deferred count marker signals that the count will be provided in the trailer. The ##! summary line provides all counts after the data is complete. The LLM has both the data and the counts in its context window (recency bias in transformer attention means the trailer is at least as strong a signal as header counts).

Streaming mode enables zero-buffering encode: rows emit the instant they are produced, with O(1) memory per row. This is critical for MCP servers that walk large graphs or paginate results; the LLM starts receiving context immediately instead of waiting for the full traversal.

TOON cannot add streaming without a breaking spec change (their grammar mandates upfront [N] with no deferred count or trailer mechanism).

4. Implementation Status

GCF is not a speculative format proposal. It is implemented in six languages, all at v1.0.0+, published to seven package registries, covered by 141 conformance fixtures, verified across 200M+ property-based round-trips and 7.9M fuzz executions, and deployed in production MCP servers.

The implementation includes:

- **Go library** (github.com/blackwell-systems/gcf-go, v1.0.2): Encode, Decode, EncodeGeneric, DecodeGeneric, EncodeWithSession, EncodeDelta, StreamEncoder, GenericStreamEncoder, ParseJSONOrdered, PackRoot. CLI with both profiles. Zero dependencies.
- **TypeScript library** (@blackwell-systems/gcf on npm, v1.0.1): encode, decode, encodeGeneric, decodeGeneric, encodeWithSession, encodeDelta, StreamEncoder, GenericStreamEncoder. CLI with both profiles. Zero dependencies, ESM.
- **Python library** (gcf-python on PyPI, v1.0.1): encode, decode, encode_generic, decode_generic, encode_with_session, encode_delta, StreamEncoder, GenericStreamEncoder. CLI with both profiles. Zero dependencies, Python 3.9+.

- **Rust library** (`gcf` on `crates.io`, v1.0.1): `encode`, `decode`, `encode_generic`, `decode_generic`, `encode_with_session`, `encode_delta`, `StreamEncoder`, `GenericStreamEncoder`. CLI with both profiles. Minimal dependencies (`serde_json`).
- **Swift library** (`gcf-swift` via SPM, v1.0.1): `encode`, `decode`, `encodeGeneric`, `decodeGeneric`, `encodeWithSession`, `encodeDelta`, `StreamEncoder`, `GenericStreamEncoder`, `OrderedDictionary`. CLI with both profiles. 50M property-based round-trips. Zero dependencies.
- **Kotlin library** (`gcf-kotlin` via JitPack, v1.0.1): `encode`, `decode`, `encodeGeneric`, `decodeGeneric`, `encodeWithSession`, `encodeDelta`, `StreamEncoder`, `GenericStreamEncoder`. CLI with both profiles. Zero dependencies.
- **MCP proxy** (`github.com/blackwell-systems/gcf-proxy`, v0.10.0): bidirectional translation (JSON-to-GCF responses, GCF-to-JSON requests), session dedup (40% savings proven e2e), proxy-level delta encoding (68% savings on incremental changes), HTTP backend (`--upstream`), HTTP/SSE frontend (`--http`), response caching (`--cache`), min-size bypass (default 100 bytes), streaming progress notifications. Zero code changes to upstream.
- **Cross-language matrix**: 5x5 encode/decode matrix verified (all passing). Each language's encoder output decoded by every other language's decoder.
- **Conformance test suite** (141 v2 fixtures across both profiles): language-agnostic JSON fixtures validating encode, decode, session, delta, generic, streaming, and normative error cases.
- **Specification** (`gcformat.com`, v2.0 Stable): RFC 2119 keywords, conformance checklists, decoder error taxonomy, streaming extension, security considerations, i18n, status lifecycle.

Correctness Validation

All six implementations are tested against round-trip invariants and the shared conformance suite. A graph response encoded as GCF and decoded back must preserve node identity, kind, score, provenance, group membership, edge direction, edge type, and optional status metadata. The generic profile is validated against additional fixtures covering flat arrays, nested objects, value formatting, primitive array inlining, and edge cases.

- **200M+ property-based round-trips** across all languages (random + adversarial value generators)
- **7.9M fuzz executions** (Go native fuzzing) finding and fixing 3 bugs
- **141 conformance fixtures** across both profiles, streaming, and normative error cases
- **5x5 cross-language encode/decode matrix** verified (each encoder's output decoded by every other decoder)
- **JSON Schema validation** works on decoded output unchanged (`gcformat.com/guide/schema-validation`)

Production Deployment

- **knowing** (28 MCP tools): GCF as primary output format for code intelligence, with session deduplication and delta encoding.
- **agent-lsp** (66 MCP tools): GCF graph profile output for symbol-returning tools (blast_radius, find_callers, explore_symbol, find_references, type_hierarchy, list_symbols, find_symbol, cross_repo). First independent production consumer.
- **NeuroNest** (NETGVai): first independent commercial adoption. GCF encoder in tool executor, swarm coordinator, and MCP server manager.
- **gcf-proxy session dedup** proven end-to-end: 40% savings on repeated queries, compounding across a session. Tested against agent-lsp on real TypeScript codebase.

5. Where Token Savings Come From

The savings decompose into five sources:

Source	JSON cost	GCF cost	Savings per occurrence
Field names	9 field names x ~2 tokens each = ~18 tokens/symbol	0 (positional)	~18 tokens/symbol
Edge references	2 qualified names x ~15 tokens = ~30 tokens/edge	2 local IDs x ~1 token = ~2 tokens/edge	~28 tokens/edge
Structural delimiters	{, }, [,], :, ", " = ~6 tokens/symbol	0	~6 tokens/symbol
Distance fields	"distance": N = ~3 tokens/symbol	0 (implicit in group)	~3 tokens/symbol
Kind strings	"function" = ~2 tokens	fn = 1 token	~1 token/symbol

For a 10-symbol, 8-edge payload: JSON ~965 tokens, GCF ~233 tokens. The 732-token difference breaks down roughly as: 280 from field name elimination, 224 from edge reference compression, 60 from delimiter removal, 30 from group headers, and the remainder from kind abbreviations and whitespace.

5.1 Generic Profile Savings

The generic profile achieves savings through a subset of the same mechanisms:

Source	JSON cost	GCF cost	Savings per occurrence
Field names	repeated per record	declared once in header	$\sim(N-1) \times$ fields per array
Structural delimiters	{, }, :, ,, " per record	between values	~ 6 tokens/record
Array framing	[,], commas	[count] in header	fixed
Primitive arrays	["a","b","c"] with brackets and quotes	name[3]: a,b,c	$\sim 50\%$ per array
Nesting	braces + field names	.field {} + key=value	$\sim 50\%$ per nested object

For 2,000 employee records with 6 fields: JSON $\sim 127,050$ tokens, GCF $\sim 49,055$ tokens (61% savings). On TOON's benchmark, GCF's generic profile uses 34% fewer tokens than TOON on mixed-structure data and wins all 6 datasets.

6. Benchmarks

All benchmarks encode the same semantic content in JSON and GCF. Token counts in Section 6.1 use cl100k_base; the 500-symbol eval (Section 6.1a) and TOON comparison (Section 8.5) use o200k_base (matching TOON's benchmark methodology). Measurements taken on Apple M4 Pro.

6.1 Token Comparison

Payload	Symbols	Edges	JSON tokens	GCF tokens	Savings
context_for_task	10	8	965	233	75.9%
context_for_task (large)	30	24	2,968	649	78.1%
context_for_files	15	12	1,490	334	77.6%
blast_radius	8	6	835	208	75.1%
semantic_diff	12	10	1,206	295	75.5%
graph_query	20	16	2,078	423	79.6%

Median token savings: 76.7%

Savings increase with payload size because the ratio of edge references to node declarations grows, and edge references are where compression is strongest.

6.1a Comprehension Accuracy at Scale (500 symbols, multi-model)

23 comprehension runs across 10 models and 3 providers. 500 symbols, 200 edges, 13 structured extraction questions with deterministic ground truth (no LLM judge). Each run generates a fresh random payload. Zero format instructions in the prompt.

Model	Runs	GCF avg	TOON avg	JSON avg
Claude Opus 4.6	2	96.2%	84.6%	73.1%
Claude Sonnet 4.6	2	100%	73.1%	53.8%
Claude Haiku 4.5	2	96.2%	69.2%	57.7%
GPT-5.5	5	84.1%	67.7%	45.8%
GPT-5.4	4	78.0%	56.0%	44.1%
GPT-5.4-mini	2	71.8%	64.1%	54.2%
Gemini 2.5 Pro	1	100%	76.9%	58.3%
Gemini 3.1 Pro	1	100%	76.9%	46.2%
Gemini 3.5 Flash	1	100%	61.5%	46.2%
Gemini 2.5 Flash	3	80.6%	54.6%	57.0%

GCF wins 22 of 23 runs (1 tie, 0 losses). Four models achieve 100%.

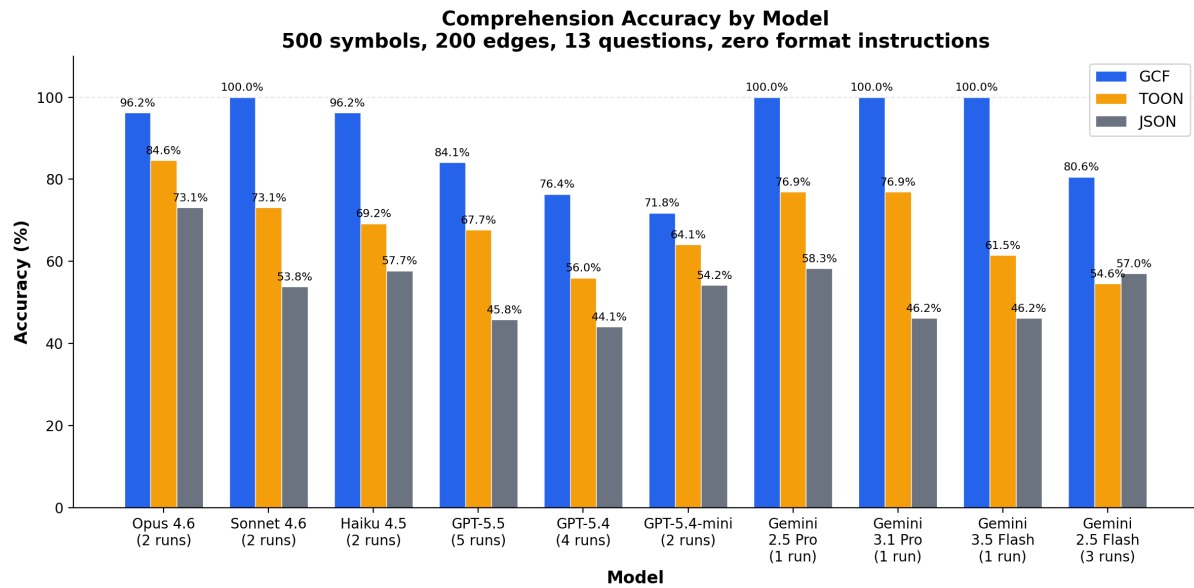


Figure 2: Comprehension Accuracy by Model

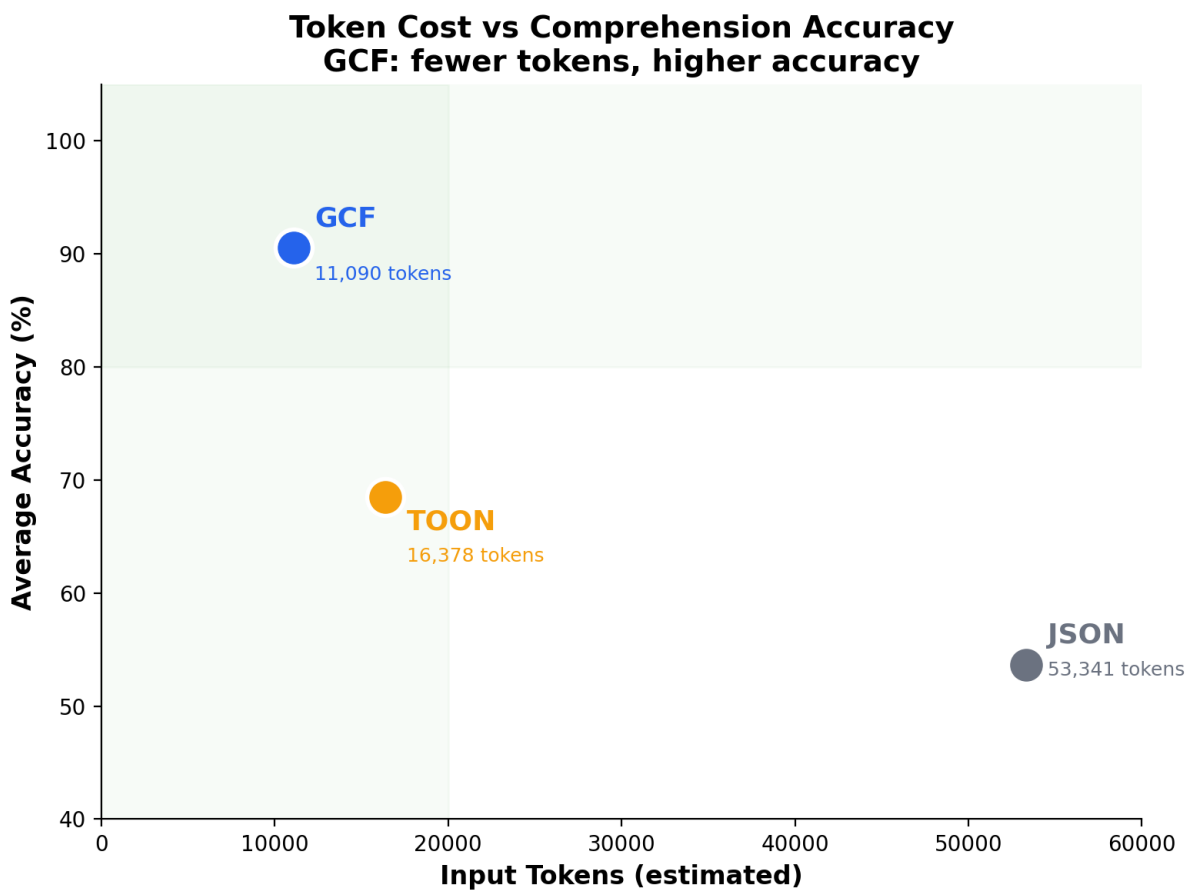


Figure 3: Token Cost vs Comprehension Accuracy

Failure taxonomy GCF, TOON, and JSON produce qualitatively different failure modes:

GCF fails on precision (median error: 4). Off-by-1 header misreads (5 occurrences), deterministic column scan miscounts on GPT-5.4 (10), and context overwhelm empty responses on GPT-5.5 (10). The format structure is understood; the count is slightly misread.

TOON fails on comprehension (median error: 53). Distance grouping failures across all models (25 occurrences), round-number guessing where the model gives up counting (7), attention decay on the last row (5), and context overwhelm (20). The model cannot filter a flat 500-row table by column value.

JSON fails on structural overwhelm (median error: 56). Empty string responses where the model produces nothing (33), massive undercounts (9), distance filter failures (29), and field confusion (3). At 53,000 tokens of repeated field names, the format itself prevents comprehension.

On two separate runs, Claude Opus responded to a JSON counting question by manually enumerating symbols one by one (143 lines on run 1, 119 on run 2), burning output tokens on a chain-of-thought enumeration and still getting the wrong answer. GCF answers the same question from a 3-character header: [167].

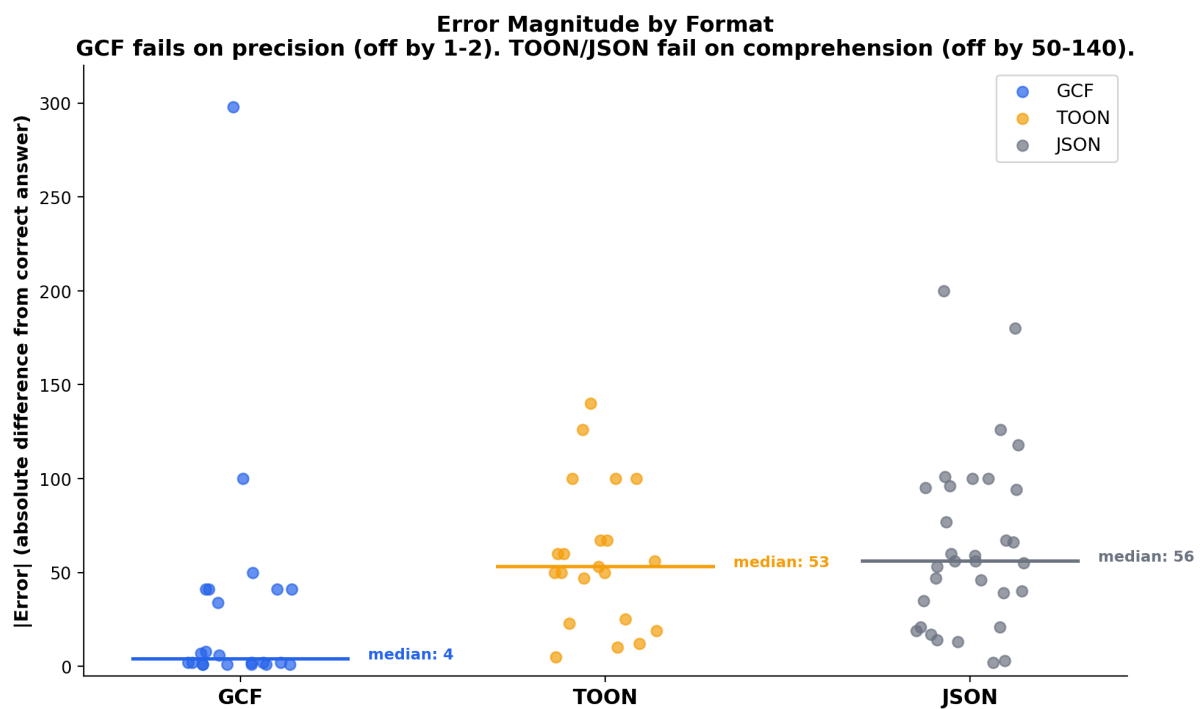
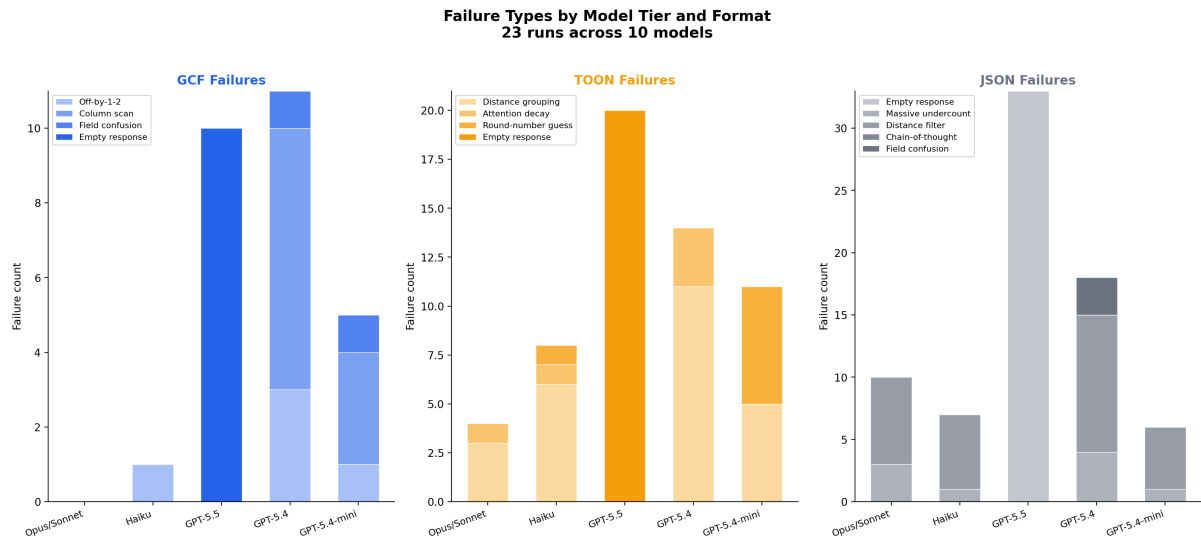
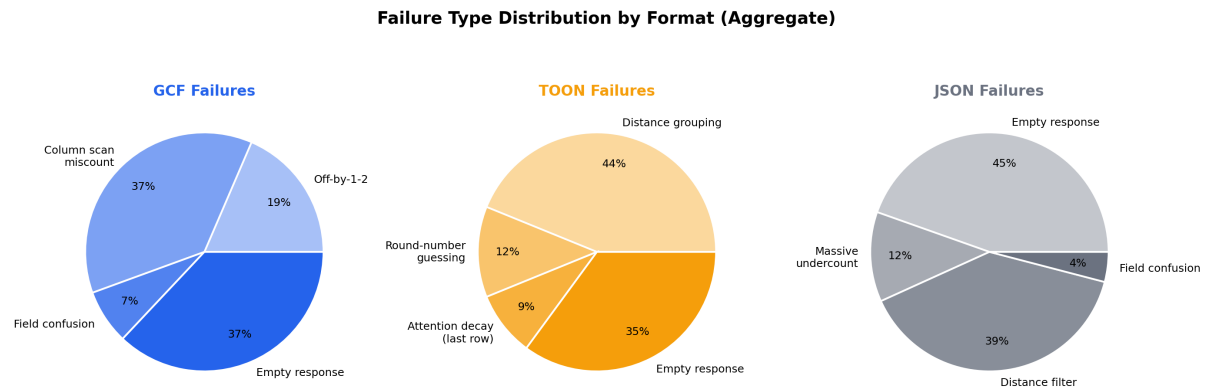
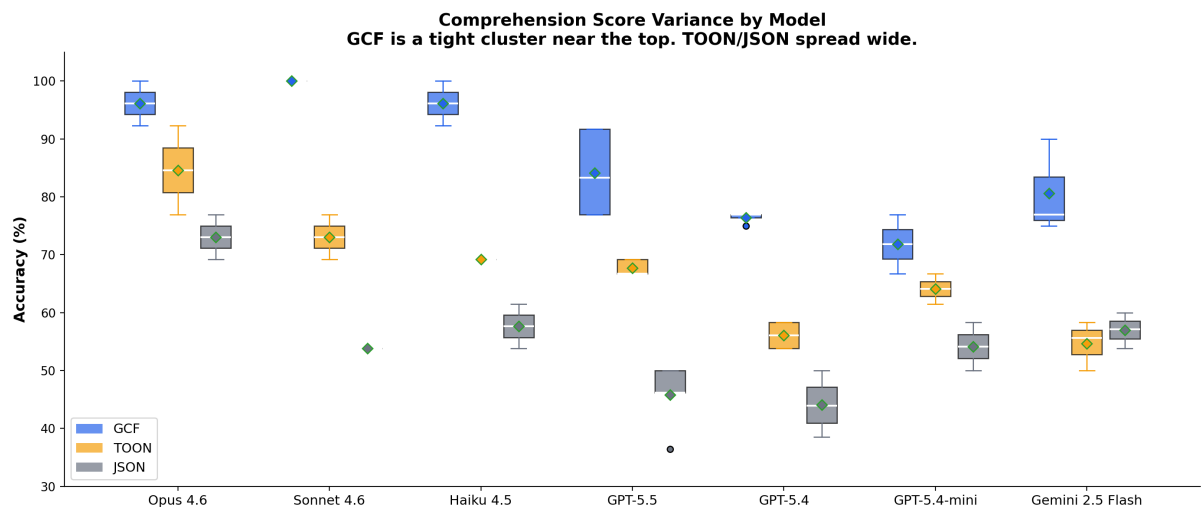


Figure 4: Error Magnitude by Format

**Figure 5: Failure Types by Model Tier****Figure 6: Failure Type Distribution****Figure 7: Comprehension Score Variance**

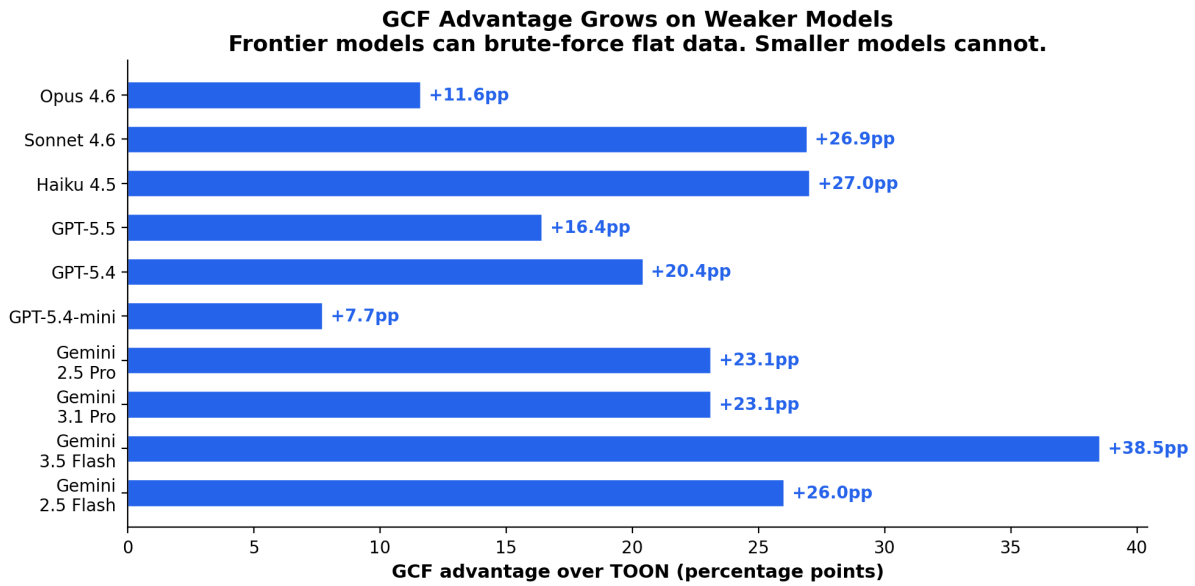


Figure 8: GCF Advantage Grows on Weaker Models

Comprehension methodology

- OpenAI runs used default temperature (non-zero). EVAL_TEMPERATURE=0 available for deterministic runs.
- Claude evals via `claude -p CLI` with `--model` flag.
- OpenAI evals via chat completions API with exponential backoff on 429s.
- Google evals via generativelanguage API with retry logic.
- TOON encoding uses the official `toon-go` library from the `toon-format` GitHub organization.
- All raw logs in `eval/results/comprehension/`.

6.2 Session Statefulness Savings

Call	New symbols	Reused symbols	GCF tokens	Cumulative savings vs JSON
1	10	0	233	75.9%
2	5	4	128	82.3%
3	3	6	87	86.1%
4	2	7	62	89.4%
5	1	8	41	92.7%

By the fifth tool call in a session, GCF achieves 92.7% token savings versus JSON because 8 of 9 referenced symbols are bare ID references (`@0`, `@3`) consuming 1 token each instead of 15-20 tokens for the full qualified name.

7. Comparison to Alternatives

7.1 Columnar/TSV

Column headers with tab-separated values eliminate field name repetition, achieving ~27% token savings. But TSV treats graph data as flat tables: edge references still require full identifier strings in each row. TSV cannot exploit referential identity (local IDs) or topological encoding (edge arrows). GCF triples the savings.

7.2 Binary Formats (Protobuf, MessagePack, FlatBuffers)

These optimize for machine parsing and byte size. An LLM cannot read a protobuf payload; it must be decoded to text first, and the decoded text is typically JSON, eliminating the savings at the point of consumption. Binary formats solve a different problem (machine efficiency) than GCF (LLM token efficiency).

7.3 JSON-LD / RDF

Verbose by design: full URIs, type annotations, @context declarations. Optimizes for semantic interoperability across systems, not token-constrained consumption. JSON-LD is worse than JSON for LLM tool responses.

7.4 Markdown / Freeform Text

Some tools return markdown-formatted text. This is human-optimized, not LLM-optimized. Markdown uses whitespace for structure (indentation, line breaks, headers) which tokenizers handle inconsistently. GCF's positional format produces consistent, predictable token counts regardless of tokenizer implementation.

7.5 TOON (Token-Oriented Object Notation)

TOON is a tabular encoding format for JSON that declares array fields once and uses comma-separated rows. It achieves 30-60% savings versus JSON on flat tabular data.

TOON:

```

tool: example
symbols[3]{name,kind,score,distance}:
  pkg.Auth,function,0.9,0
  pkg.Server,function,0.7,1
  pkg.Cache,type,0.4,2
edges[1]{source,target,type}:
  pkg.Server,pkg.Auth,calls

```

GCF (same data):

```

GCF profile=graph tool=example symbols=3 edges=1
## targets
@0 fn pkg.Auth 0.90 lsp
## related
@1 fn pkg.Server 0.70 lsp
## extended
@2 type pkg.Cache 0.40 structural
## edges [1]
@0<@1 calls

```

The structural difference: TOON puts all symbols in one flat table with a distance column. GCF groups them into `## targets`, `## related`, `## extended` sections. This difference drives both the comprehension gap (models can't filter flat tables at scale) and the generation gap (models write "target" instead of 0 in TOON's distance column).

Methodology: We forked TOON's benchmark repository (github.com/toon-format/toon), added GCF as one additional formatter (a single file importing `encodeGeneric` from the published `@blackwell-systems/gcf` npm package), and ran their benchmark harness unchanged. Datasets, tokenizer (gpt-tokenizer, o200k_base), and methodology are entirely upstream. Results:

Dataset	GCF	TOON	Result
Semi-uniform event logs	108,158	154,032	GCF 42% smaller
E-commerce orders	61,593	73,246	GCF 19% smaller
Employee records (flat)	49,055	49,966	GCF 2% smaller
Analytics time-series	8,398	9,127	GCF 8% smaller
GitHub repos	8,576	8,744	GCF 2% smaller
Deeply nested config	616	618	GCF 0.3% smaller

GCF wins all 6 datasets. TOON has no token efficiency advantage on any data shape.

TOON has no local-ID system, no edge encoding, no session deduplication, no delta encoding, and no streaming mode. These are structural limitations that cannot be added without a fundamental redesign. TOON edges must repeat the full qualified name of both source and target (~100 tokens per edge vs ~4 for GCF). TOON retransmits every record on every call; GCF tracks what's been sent. TOON's spec mandates upfront [N] counts with no deferred mechanism; GCF streams with zero buffering.

TOON is fundamentally fragile for generation. Its official decoder rejects LLM-generated output on 7 of 9 models tested. The failure is always the same: models write semantic labels where TOON expects integers. This is a structural design flaw in flat tabular formats that encode categories as column values. GCF solves this by expressing categories through section placement. See Section 9.1 for the full analysis.

On output, GCF produces 63% smaller output than JSON and 33% smaller than TOON at 100 symbols.

Fork with reproducible results: github.com/blackwell-systems/toon (branch: `gcf-comparison`).

7.6 Custom Compressed JSON

JSON with shortened field names (`"qn"` instead of `"qualified_name"`) achieves 15-25% savings. This preserves JSON's structural overhead (delimiters, quoting, nesting) while reducing readability. GCF eliminates the structural overhead entirely rather than shortening the labels on it.

8. Limitations and Validation

LLM comprehension. 23 runs across 10 models and 3 providers validate this design. GCF averages 90.7% accuracy where JSON averages 53.6% and TOON averages 68.5%. Four models achieve 100% on GCF. The concern that LLMs might struggle with GCF's dense positional format was unfounded; the format improves comprehension by eliminating structural noise. GCF's @N local IDs, ## section headers, and | pipe separators are more comprehensible to an LLM than JSON's `"qualified_name"`: repeated 500 times.

LLM generation. 28 generation runs across 9 models and 3 providers. GCF achieves 5/5 validity on every frontier model (Opus, Sonnet, GPT-5.5, Gemini 2.5 Pro, Gemini 3.1 Pro) with a 3-line primer and zero prior training. TOON's official decoder rejects LLM-generated output on 7 of 9 models. The failure is structural: TOON's flat columns encode semantic categories as integers, and models write labels (`"target"`) instead of the expected integer (0). GCF expresses categories through section placement (`## targets`), aligning with how models naturally express grouped data. GCF output is 63% smaller than JSON and 33% smaller than TOON.

Two profiles. The graph profile is optimized for typed nodes and edges. The generic profile (`encodeGeneric`) handles arbitrary structured data (arrays of objects, nested records, mixed

types). On TOON's own benchmark with their datasets and tokenizer, GCF's generic profile wins all 6 datasets. Primitive array inlining (name[N]: val1, val2, val3) eliminated TOON's single remaining advantage on deeply nested config.

Session statefulness requires coordination. The session ID system assumes the server tracks which nodes have been transmitted to which client. Stateless deployments cannot use session compression. The non-session mode (full retransmission) remains available.

Tokenizer dependency. Token counts depend on the tokenizer. The benchmarks in this paper use o200k_base. Different tokenizers produce different token counts for the same GCF payload, though the relative savings versus JSON are consistent (within 2-3 percentage points).

9. Implications for MCP and Agent Tooling

The MCP specification does not define a standard for tool response encoding beyond “the response is a JSON-RPC result.” This means every MCP server independently decides how to format its output. The result is that agents receive verbose JSON from every tool, with no mechanism to request compact encoding.

We propose that MCP tool responses should support format negotiation: the client specifies a preferred encoding in the tool call, and the server returns the response in that encoding. GCF (or a format like it) should be available as a standard option alongside JSON.

The token savings are too large to ignore. A 79% reduction in tool response tokens translates directly to: lower API costs, faster time-to-first-token, more room in the context window for source code and reasoning, and fewer multi-turn loops caused by context window exhaustion.

9.1 LLM Generation: GCF as a Bidirectional Format

GCF is bidirectional: LLMs can produce it, not just consume it. 28 generation runs across 9 models and 3 providers, validated through real decoders (including TOON's official toon-go library).

Model	GCF	TOON (natural)	JSON
Claude Opus 4.6	5/5	0/5	5/5
Claude Sonnet 4.6	5/5	2-3/5	5/5
Claude Haiku 4.5	5/5	1-3/5	5/5
GPT-5.5	4-5/5	1-2/5	5/5

Model	GCF	TOON (natural)	JSON
GPT-5.4	5/5	0/5	5/5
GPT-5.4-mini	5/5	0/5	5/5
Gemini 2.5 Pro	5/5	1/5	5/5
Gemini 3.1 Pro	5/5	0/5	5/5
Gemini 3.1 Flash Lite	4-5/5	0/5	4-5/5

GCF 5/5 on every frontier model. TOON fails on 7 of 9 models.

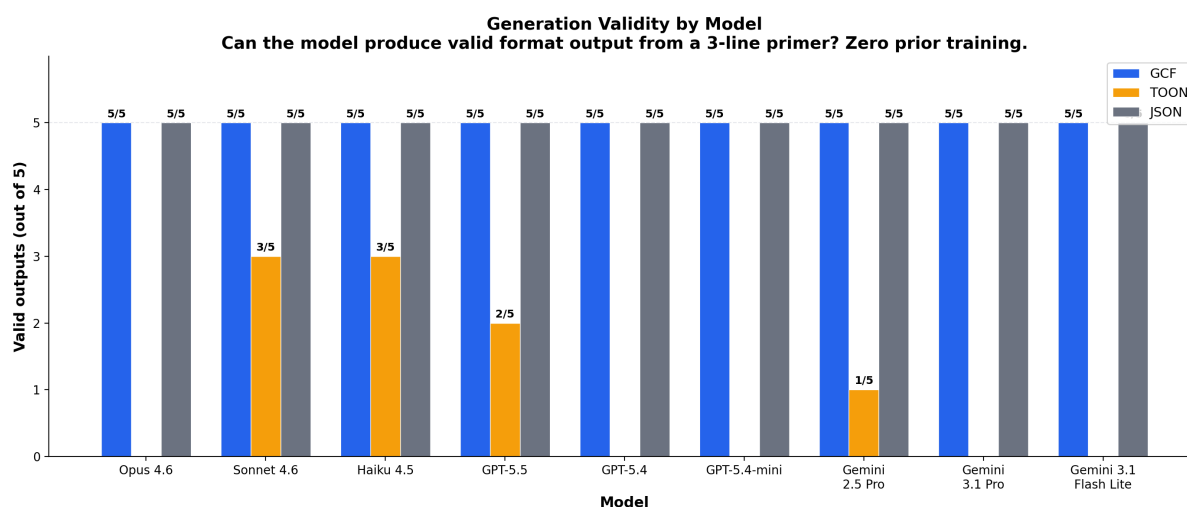


Figure 9: Generation Validity by Model

Why TOON fails generation TOON’s flat tabular design encodes semantic categories as column values. When told “this symbol is a target,” the model writes target in the distance column. TOON’s decoder expects 0. The model would need to know, unprompted, that “target” maps to 0, “related” maps to 1, “extended” maps to 2. No model does this. The error is always the same: `toon: cannot assign string to int`.

This is a structural design flaw in flat tabular formats. Any time a column encodes a semantic category as an integer, the format is one prompt change away from producing invalid data. TOON is fundamentally fragile for LLM generation.

GCF expresses categories through section placement (`## targets`, `## related`). The model writes the symbol in the section matching the label. No integer mapping required. The format aligns with how LLMs naturally express grouped data.

The Distance Label Problem

Why TOON fails generation on 5 of 7 models

GCF: Model writes symbol in matching section	TOON: Model writes labels, decoder expects integers
## targets	symbols[3]{name,kind,distance}:
@0 fn pkg.Foo 0.95 lsp	pkg.Foo,fn,target expected: 0
## related	pkg.Bar,type,related expected: 1
@1 type pkg.Bar 0.70 ast	pkg.Baz,iface,extended expected: 2
## extended	
@2 iface pkg.Baz 0.30 lsp	
Distance is structural. No mapping needed.	Model writes "target" not 0. TOON decoder rejects it.

Figure 10: The Distance Label Problem

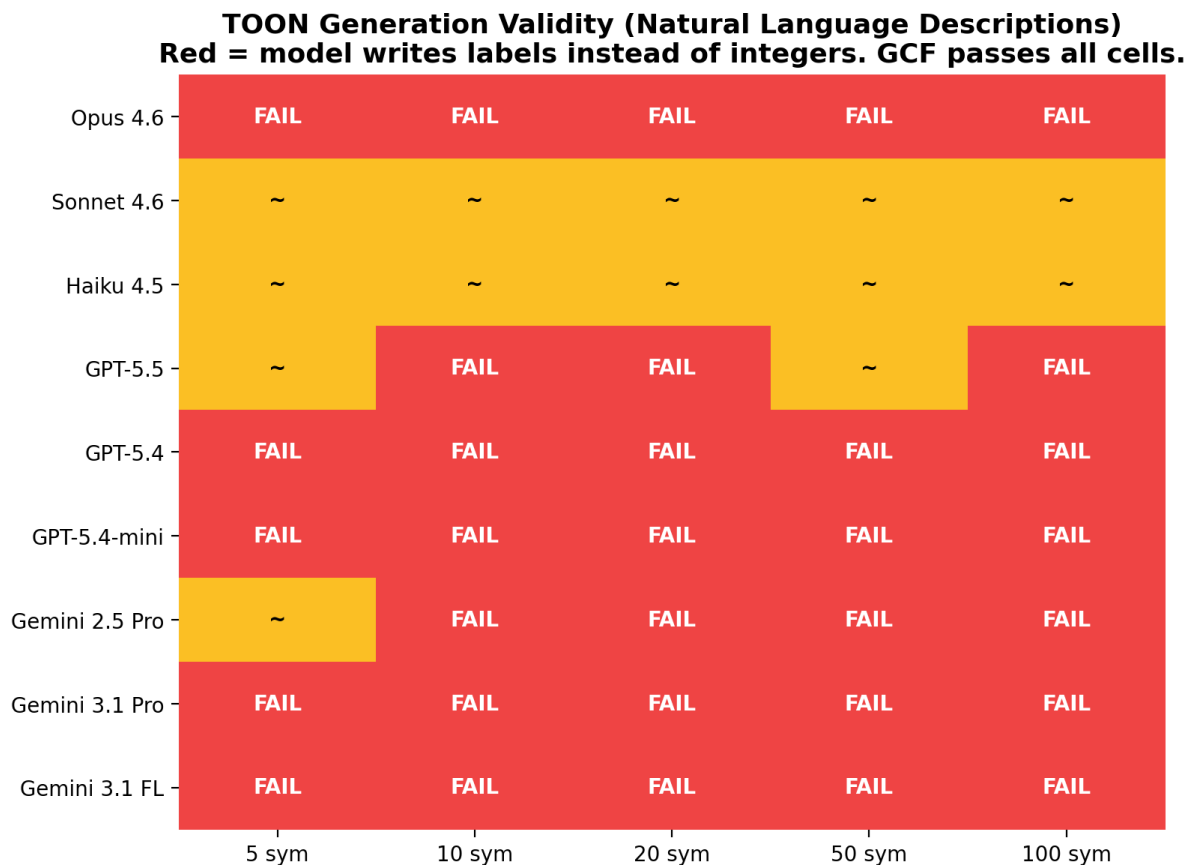


Figure 11: TOON Generation Heatmap

When TOON is given pre-encoded integers (hand-holding the model through the mapping), performance improves on some models but remains inconsistent. Even in the best case, GCF output is 28% smaller.

Output size comparison

Format	100 sym output	vs JSON
GCF	5,976 B	63% fewer
TOON	8,937 B	45% fewer
JSON	16,121 B	baseline

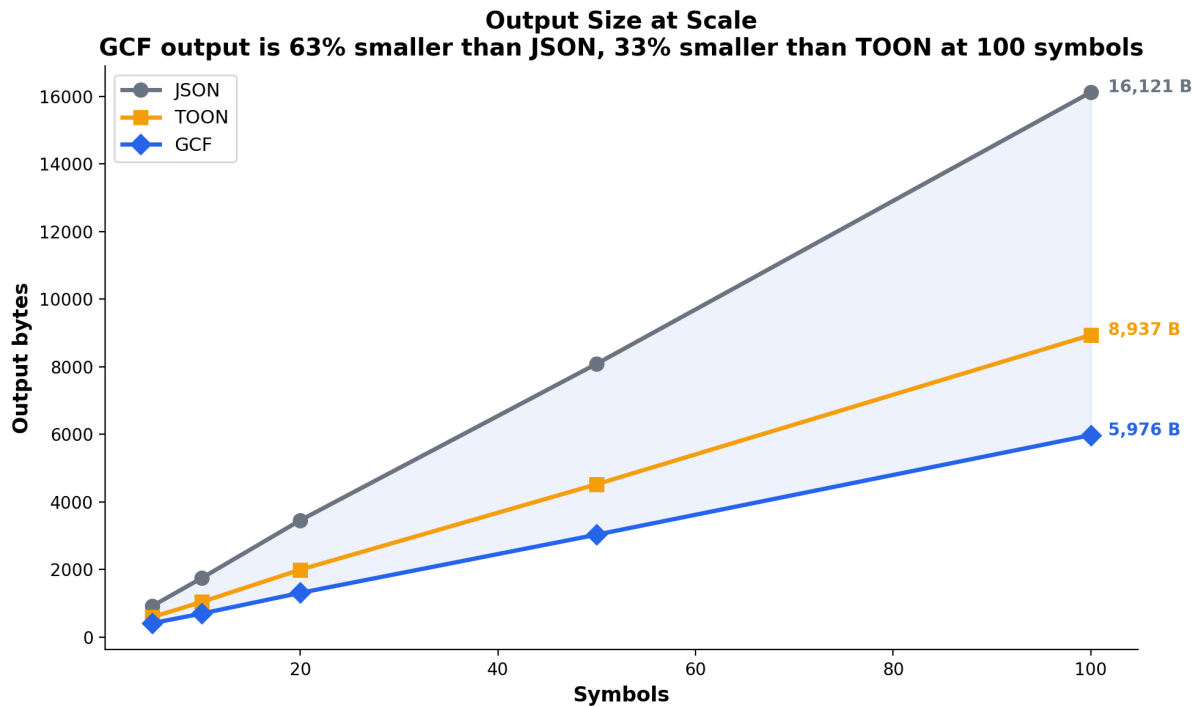


Figure 12: Output Size at Scale

Implications. GCF generation enables: (1) agent-to-agent communication at 63% fewer tokens per handoff, (2) structured output mode as an alternative to JSON mode, and (3) system prompt encoding where context packs are encoded as GCF to maximize context window utilization.

9.2 Streaming: Progressive Context Delivery

The streaming encoding extension (Section 3.8) enables a new interaction pattern: **progressive context delivery**. An MCP proxy can emit GCF fragments as MCP progress notifications while the upstream server is still processing. The LLM receives partial context immediately, reducing perceived latency from seconds to milliseconds on large graph traversals.

Combined with MCP's Streamable HTTP transport (Server-Sent Events), this creates a pipeline where graph data flows from the server through the proxy to the LLM token by token, with the trailer summary providing count verification at the end.

9.3 Delta Encoding

GCF's token savings compound with delta encoding: when the agent passes a `pack_root` from a prior call and the pack changed, the server sends only added/removed symbols instead of the full payload. Measured: 81.2% additional token savings at 96.6% symbol overlap on re-query scenarios. Combined with GCF's baseline savings and session deduplication, the three-level stack achieves over 97% cumulative token reduction on warm sessions versus stateless JSON.

10. Conclusion

JSON is the default encoding for LLM interactions because it is universal, not because it is efficient. For structured data, JSON wastes more than three-quarters of its tokens on structural overhead that carries no semantic content.

GCF eliminates this waste through two encoding profiles. The graph profile uses referential identity (local IDs), topological encoding (edge arrows), and hierarchical grouping (section headers) for code graph data. The generic profile uses positional rows with pipe separators, section headers, and inline primitive arrays for arbitrary structured data. Both achieve significant savings: 79% versus JSON on graph data at 500 symbols, 34% versus TOON on TOON's own mixed-structure benchmark (winning all 6 datasets).

GCF is bidirectional. 1,300+ LLM evaluations across 10 models and 3 providers prove it. Comprehension: 90.7% average accuracy (four models at 100%) where JSON averages 53.6% and TOON averages 68.5%. Generation: 5/5 validity on every frontier model where TOON fails on 7 of 9 models. Output: 63% fewer tokens than JSON, 33% fewer than TOON. Session deduplication (92.7% by the fifth call), delta encoding (81.2% on re-queries), and streaming encode (zero-buffering with trailer summary) compound savings across multi-turn interactions. No competing format offers these features.

The format is text-based, LLM-optimized, and implementable in any language. Implementations exist in six languages (Go, TypeScript, Python, Rust, Swift, Kotlin), all at v1.0.0+ with CLIs, zero or minimal runtime dependencies, and 200M+ verified lossless round-trips. A bidirectional MCP proxy enables adoption with zero code changes: session dedup (40% savings proven e2e), proxy-level delta encoding (68% savings on incremental changes), HTTP backend/frontend for remote deployment, response caching, and streaming progress notifications. JSON Schema validation works on decoded output unchanged, eliminating the primary enterprise adoption objection.

The broader point: GCF is a wire format. Wire formats are not optimized for human readability. HTTP headers are not readable. Protobuf is not readable. Nobody cares; they use a viewer. GCF is the wire format; JSON is the viewer format. The agent reads GCF (cheap, accurate), does its work, then calls `decode()` at the end if a human needs to see the result. Human readability is a last-mile rendering concern, not a wire format property.

The format that looks clean to humans (JSON) is the one that breaks for agents at scale. The format optimized for agentic comprehension (GCF) achieves 90.7% average accuracy across 10 models at the lowest token cost, with four models hitting 100%. TOON, the format positioned as the compromise between human readability and machine efficiency, fails generation on 7 of 9 models and never achieves 100% comprehension on any model. This is not a trade-off. It is a design choice validated by 1,300+ evaluations across every major AI provider.

Reference Implementation

- **Specification:** gcfformat.com (v2.0 Stable, RFC 2119 keywords, conformance checklists, streaming extension, error taxonomy, i18n, status lifecycle)
- **Go library:** github.com/blackwell-systems/gcf-go (v1.0.2). CLI: `gcf encode-generic`, `gcf decode-generic`
- **TypeScript library:** [@blackwell-systems/gcf](https://github.com/blackwell-systems/gcf) on npm (v1.0.1). CLI: `npx @blackwell-systems/gcf encode-generic`
- **Python library:** `gcf-python` on PyPI (v1.0.1). CLI: `python -m gcf encode-generic`
- **Rust library:** `gcf` on crates.io (v1.0.1). CLI: `gcf encode-generic`
- **Swift library:** `gcf-swift` via SPM (v1.0.1). CLI: `swift run GCFCLI encode-generic`
- **Kotlin library:** `gcf-kotlin` via JitPack (v1.0.1). CLI: `gradle run --args="encode-generic"`
- **MCP proxy:** github.com/blackwell-systems/gcf-proxy (v0.10.0): bidirectional translation, session dedup, delta encoding, HTTP backend/frontend, response caching, min-size bypass, streaming progress
- **Comprehension eval:** github.com/blackwell-systems/gcf-go/eval (500 symbols, 13 questions, 3 formats, 10 models, 3 providers)
- **Generation eval:** github.com/blackwell-systems/gcf-go/eval (5-100 symbols, GCF vs TOON vs JSON, 9 models, validated through real decoders)
- **Eval results:** github.com/blackwell-systems/gcf/eval/results (all raw logs, failure taxonomy, artifacts)
- **TOON benchmark fork:** github.com/blackwell-systems/toon (branch: `gcf-comparison`, their datasets, their tokenizer)
- **Conformance test suite:** github.com/blackwell-systems/gcf/tests/conformance (141 v2 fixtures across both profiles, streaming, and normative errors)
- **Cross-language matrix:** github.com/blackwell-systems/gcf/tests/cross-language-matrix.py (5x5 encode/decode verification)
- **Schema validation guide:** gcfformat.com/guide/schema-validation (JSON Schema on decoded output, three validation patterns)
- **FAQ:** gcfformat.com/guide/faq (protobuf, MessagePack, CBOR, gzip, YAML comparisons)
- **Interactive playground:** gcfformat.com/playground (three-way JSON vs TOON vs GCF comparison using real `@toon-format/toon` library)

- **Production deployment:** knowing (28 MCP tools), agent-lsp (66 MCP tools), NeuroNest (first independent commercial adoption)

Appendix A: Full Example

JSON (965 tokens):

```
{
  "tool": "context_for_task",
  "tokens_used": 1847,
  "token_budget": 5000,
  "symbols": [
    {
      "qualified_name": "github.com/blackwell-systems/knowning/internal/mcp.requireHash",
      "kind": "function",
      "score": 0.78,
      "signature": "func requireHash(args map[string]any, key string) (types.Hash, error)",
      "provenance": "lsp_resolved",
      "distance": 0,
      "components": { "blast_radius": 0.40, "confidence": 0.25, "recency": 0.06, "distance": 0.15 }
    },
    {
      "qualified_name": "github.com/blackwell-systems/knowning/internal/mcp.NewServer",
      "kind": "function",
      "score": 0.54,
      "provenance": "lsp_resolved",
      "distance": 1
    }
  ],
  "edges": [
    {
      "source": "github.com/blackwell-systems/knowning/internal/mcp.NewServer",
      "target": "github.com/blackwell-systems/knowning/internal/mcp.requireHash",
      "edge_type": "calls"
    }
  ]
}
```

GCF (233 tokens):

```
GCF profile=graph tool=context_for_task budget=5000 tokens=1847 symbols=2 edges=1
## targets
@0 fn github.com/blackwell-systems/knowning/internal/mcp.requireHash 0.78 lsp_resolved
## related
@4 fn github.com/blackwell-systems/knowning/internal/mcp.NewServer 0.54 lsp_resolved
## edges [1]
@0<@4 calls
```

Same semantic content. 75.9% fewer tokens.

Appendix B: Streaming Example

Buffered mode (full payload known upfront):

```
GCF profile=graph tool=context_for_task budget=5000 symbols=3 edges=2
## targets
@0 fn pkg.Auth 0.95 lsp_resolved
@1 fn pkg.Handler 0.88 lsp_resolved
## related
@2 type pkg.Config 0.72 ast_inferred
## edges [2]
@0<@1 calls
@2<@0 references
```

Streaming mode (data arriving incrementally):

```
GCF profile=graph tool=context_for_task budget=5000
## targets
@0 fn pkg.Auth 0.95 lsp_resolved
@1 fn pkg.Handler 0.88 lsp_resolved
## related
@2 type pkg.Config 0.72 ast_inferred
## edges [?]
@0<@1 calls
@2<@0 references
##! summary counts=2
```

Both produce identical Payload structures when decoded. The streaming mode enables zero-buffering encode with $O(1)$ memory per row.