An alternative to JSON responses.

Argo at WhatsApp

Kevin Gorham

gmale.github.io

GraphQLConf

2026

story

WhatsApp as a utility.

users · businesses · countries

Somewhere right now,

a user's request is failing

because it's **30% bigger** than it needs to be.

quick history

XML → WAP → GraphQL/JSON → ?

```
<message to="5551234567@whatsapp.net"
         from="5559876543@whatsapp.net"
         id="msg_001"
         type="chat">
  <body>Hi</body>
  <request xmlns="urn:xmpp:receipts"/>
</message>
```

F8 0A B3 A2 FA 04 35 35 35 31 32 33 34 35 36 37
01 A5 FA 04 35 35 35 39 38 37 36 35 34 33 01 8A
FC 07 6D 73 67 5F 30 30 31 AB 9C F8 04 76 00 FC
03 48 69 2E F8 03 D5 A0 DB 00

30% larger

JSON responses vs WAP-encoded equivalents

WAP.

the formidable baseline

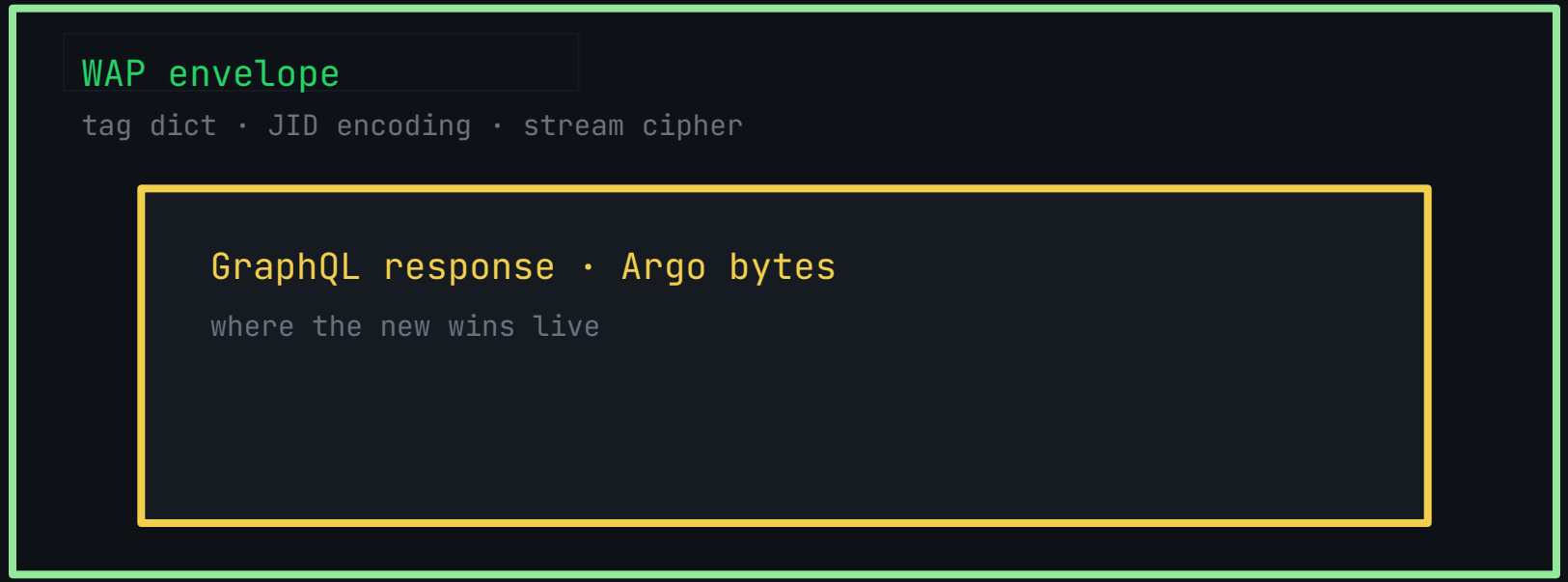
WAP is not naive

Already does most of what you'd want.

- Tag dictionary: common XML tags become byte indices
- Custom JID encoding: userID@s.whatsapp.net → a few bytes
- Stream cipher: encryption inline
- Versioned protocol: currently v6, evolves over time

the wrap

Our payload lives inside WAP.



Argo isn't replacing WAP. It's competing for the delta inside.

Why not proto?

or even just gzip?

the obvious answers we didn't pick

Three reasons.

- Loses schema-awareness: proto doesn't know GraphQL.
- Not materially better than tuned WAP on small payloads.
- Doesn't model GraphQL's null/error/partial-data semantics.

We had a better tool. We had a schema.

Argo.

the format that knows the schema

argo, in one breath

A binary serialization format for GraphQL responses.

Designed by Mike Solomon. GraphQLConf 2023. github.com/msolomon/argo

front matter

---
compact: use compact binary encoding for graphql types
compressible: use compressible layout to complement compression, not compete
---

- Don’t repeat known structure
- Don’t repeat types
- Make values compact
- Generate type descriptors (wire types)

implementation

Key Components.

structure

Argo Message.

```
[header byte] [block 1] [block 2] [core]
```

under the hood

Deduplication.

JSON (repeated strings)

```
{
  "user": {
    "accountType": "STUDENT",
    "name": "Alice",
    "friends": [
      {
        "accountType": "STUDENT",
        "name": "Bob"
      },
      {
        "accountType": "STUDENT",
        "name": "Carol"
      }
    ]
  }
}
```

Argo (deduplicated)

```
STRING BLOCK:
  [0] "STUDENT"
  [1] "Alice"
  [2] "Bob"
  [3] "Carol"

CORE (skeleton):
  typename: ref(0)  ← "STUDENT"
  name:     ref(1)  ← "Alice"
  friends:
    typename: ref(0)  ← reused!
    name:     new(2)  ← "Bob"
    typename: ref(0)  ← reused!
    name:     new(3)  ← "Carol"
```

[block 1]

Repeated strings written once. Backreferences replace copies.

step 1 — walk the schema

```
%% derive the shape from the schema
WireType = argo_typer:derive_wire_type(
    ServiceDocument,
    ExecutableDocument,
    OperationName
),

%% bind values to the shape
ArgoValue = argo_value:from_json(WireType, JSONValue),

%% serialize
ArgoBytes = argo_value:to_writer(ArgoValue).
```

Walk the schema for this operation. Derive the wire-type tree.

step 2 — bind values to the shape

```
%% derive the shape from the schema
WireType = argo_typer:derive_wire_type(
    ServiceDocument,
    ExecutableDocument,
    OperationName
),

%% bind values to the shape
ArgoValue = argo_value:from_json(WireType, JSONValue),

%% serialize
ArgoBytes = argo_value:to_writer(ArgoValue).
```

Take the GraphQL response. Bind it to the wire-type tree.

step 3 — serialize

```
%% derive the shape from the schema
WireType = argo_typer:derive_wire_type(
    ServiceDocument,
    ExecutableDocument,
    OperationName
),

%% bind values to the shape
ArgoValue = argo_value:from_json(WireType, JSONValue),

%% serialize
ArgoBytes = argo_value:to_writer(ArgoValue).
```

Walk the value. Emit bytes. No keys.

Built.

ports, not a rewrite

andrew bennett, GPT

Erlang. · Kotlin. · Swift.

One engineer. ~500 lines per port.

Andrew co-designs the spec with Mike Solomon.

What does a feature engineer see?

Nothing.

They write .graphql files.

Responses arrive as JSON.

Argo is invisible.

Rollout.

fine-grained control.

```
{argo_blocklist, [
    <<"ChannelSubscriberMutation">>
]}.

{argo_allowlist, [
    <<"All Operations">>
]}.
```

Server-side. Per-operation.

~40 min to propagate. Client-side approach take ~24 hours.

case study

Eight days. Rest of rollout never paused.

— *rest of rollout, uninterrupted* —

Results.

predicted vs actual

STANZA CLASS	PREDICTED	P50	P90	ACTUAL AVG
< 1 KB	30%	25%	40%	**27%**
> 1 KB	5%	5%	16%	**9%**

the aha moment

When the answer is "nothing," Argo says it in bytes.

JSON

```
{
  "data": {
    "userAccount": {
      "pendingMessages": {
        "drafts": []
      }
    }
  }
}
```

58 bytes

Argo

00

~7 bytes

*** gzipped is larger! 73 bytes*

No need for server to send structural info to the client. ***~90% saved.***

Because the exact query is known

30% larger → 5% smaller

Argo vs WAP-encoded IQs · Performance neutral

App size under budget: 28KB Android · 70KB iOS

Prove the absence of errors.

— positive signals, not just alerts —

Next.

— **Custom binary scalar.** No more base64. Save ~33%.

— **Remain binary E2E.** Zero-copy decode. Point to Argo buffer.

— **Spec evolution.** @stream, @defer, fragment spreads.

adoption

Cost vs. Value

Argo is unlike gzip. It requires coordination.

THE COST

✓ **Server encoder.** ~1,000 lines in your server language. Erlang, TypeScript, GoLang (Beeper) exist. Others: port.

Build infra. A step that derives wire types from schema + operations, bundled into the app.

Client decoder. Requires Argo store. ~28KB Android, ~70KB iOS. Same ~1,000 lines.

Kill switch. Per-operation server-side disable. Non-negotiable. Positive signals also recommended.

Debug tooling. Binary isn't readable. You need a trace filter. We built ours after shipping. *Learn from us.*

THE VALUE

27% avg wire reduction on small payloads. p90: 40%.

>80% on empty responses — enforcements, health checks, status polls.

Base64 eliminated with custom binary scalar extra ~33% saved on binary-heavy ops.

Neutral regression. No user-facing performance regression. Wire savings are free.

~~6 months, small team~~ → **hours, one engineer + AI.** The codec is 1K lines. The patterns are documented. The mechanical work is automatable.

Key Takeaways.

— The fastest bytes are the ones you never send

- Argo was able to out-perform WAP which was fine-tuned over 10 years
- WAP, gzip, zstd all operate on what's in the payload
- We've always known smaller payloads help users on constrained networks.
- Compression makes data smaller. Argo makes data absent.

— The ROI changed. The users didn't.

- We used to say, "The ROI doesn't justify six engineer-months." That was fair.
- With AI, that barrier is now gone but the insight remains.
- The question is shifting from "can we afford to?" to "can we afford not to?"
- You come to work each day on behalf of someone who will never know your name but is counting on what you ship.
- Every byte you shave is a bet that their message gets through.

— Serve the user you'll never meet.

- GraphQL allows you to remove the information both sides already know
- A single message from an oil rig reached shore and made a difference in someone's life.
- Everything we've talked about today boils down to one question: who are you building for?

Questions?

Kevin Gorham

gmale.github.io