Conf42 Golang 2026

Building an MCP Server in Go for a Real B2B Workflow

Go

MCP

GraphQL

AWS AppSync

Securely connecting Claude to a million-company B2B platform through Go, GraphQL, and AWS AppSync

Shadi Elyafi

Practical lessons from building an MCP server against real business data

Most examples stop where a demo becomes interesting

Demo wrapper

- Thin API pass-through
- Broad tools
- Prompt-level guardrails
- Little visibility

Production interface

- Narrow contracts
- Read/write split
- Backend auth preserved
- Tests and logs

The harder question is what happens when the integration touches real business data, real workflows, and real operational constraints.

User requests are normalized into explicit tool calls

The MCP server never passes raw user input directly to GraphQL.

The MCP server is a contract-enforcing layer, not a GraphQL proxy

Auth stays inside the server boundary

1. Use OIDC bearer token, API key, or AWS SigV4 by environment
2. Keep credentials and signing inside the server boundary
3. Send standard GraphQL over HTTP to AWS AppSync
4. Let AppSync enforce backend authorization

AppSync point

- managed GraphQL gateway
- OIDC carries user identity
- the MCP layer inherits the access model

Nine tools implemented; eight exposed in the shipped configuration

Read-only tools

search_companies

get_company

get_companies_batch

ai_search

list_collections

get_collection_items

Exposed mutation-capable

add_to_collection

request_email_discovery

Implemented, not exposed

create_collection

Held back after real AppSync validation. Slide 15 shows why.

Read-only tools are never passed the mutation flag

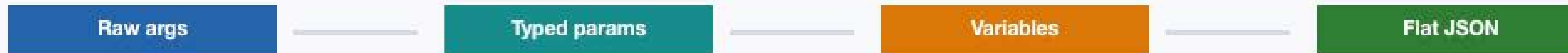
```
func NewRegistry(client graphql.Client, allowMutations bool) *Registry {
    return &Registry{
        searchCompanies: NewSearchCompaniesTool(client),
        getCompany:      NewGetCompanyTool(client),
        aiSearch:        NewAISearchTool(client),

        addToCollection: NewAddToCollectionTool(client, allowMutations),
    }
}
```

Design decision

Read-only tools have no mutation path to gate. Mutation-capable tools receive the explicit flag.

Validation, execution, and shaping are handled by distinct layers




```
func (t *SearchCompaniesTool) Execute(
    ctx context.Context,
    p SearchCompaniesParams,
) (*SearchCompaniesResult, error) {
    if err := t.validate(p); err != nil { return nil, err }
    p = t.normalize(p)
    variables := t.buildVariables(p)

    var out gqlSearchCompaniesResponse
    if err := t.client.Execute(ctx, searchQuery, variables, &out); err != nil {
        return nil, err
    }
    return flattenCompanyResults(out), nil
}
```

The MCP server never passes raw user input directly to GraphQL, and never returns raw GraphQL responses to the client.

Broad queries against 1M+ profiles had to be bounded from the start

Rule	Input	Backend value
Default	limit <= 0	10
Cap	limit > 100	100
Country	DE	countries;Germany
Country	US	countries;United States

Scale context

1M+ profiles means broad inputs become confusing outputs fast.

Even with a hard limit of 100 results, broad requests can be too broad to be useful and can compound through follow-up queries.

The LLM receives a consistently shaped record without nested objects

Backend GraphQL shape

nested companyModel -> company -> description -> text

flatten

```
{
  "id": "example.com",
  "name": "Example Inc",
  "country": "Germany",
  "employeeRange": "101-250",
  "domain": "example.com",
  "industryTags": ["Software"]
}
```

Stable records reduce accidental field confusion for the LLM client.

A default-deny model makes mutation capability a conscious decision

```
func (t *AddToCollectionTool) Execute(
    ctx context.Context,
    p AddToCollectionParams,
) (*AddToCollectionResult, error) {
    if !t.mutationsAllowed {
        return nil, fmt.Errorf("mutations are disabled; use --allow-mutations")
    }

    // validation and GraphQL mutation happen only after the gate
}
```

Mutation requested

add_to_collection

Gate first

No flag -> no backend call

This is not full authorization; it is an explicit operational switch.

Mutation gating was developed test-first

```
tool := NewAddToCollectionTool(mockClient, false)

result, err := tool.Execute(ctx, AddToCollectionParams{
    CollectionID: "col-123",
    CompanyIDs: []string{"example.com"},
})

require.Error(t, err)
assert.Nil(t, result)
mockClient.AssertNotCalled(t, "Execute")
```

Critical assertion

AssertNotCalled(t, "Execute")

Failing test -> guard implementation -> passing assertion. The last assertion proves GraphQL is never called.

Go made the MCP boundary explicit and testable

```
type Client interface {
    Execute(
        ctx context.Context,
        query string,
        variables map[string]any,
        result any,
    ) error
}
```

context.Context

Cancellation and deadlines flow into AppSync calls

Small interface

Mocked tests use the same Execute contract as runtime

Typed params

Tool handlers parse raw args before GraphQL variables

Go's ordinary service patterns made the MCP layer easier to validate independently of the LLM client.

Mocked tests became valuable when they captured GraphQL variables

```
var captured map[string]any

mockClient.On("Execute", mock.Anything, mock.Anything, mock.Anything, mock.Anything).
    Run(func(args mock.Arguments) {
        captured = args.Get(2).(map[string]any)
    }).
    Return(nil)

assert.Equal(t, 100, captured["limit"])
```

What this catches

- limit capping
- country-code resolution
- backend variable shape

Mocked unit tests cannot substitute for validation against the real system

The real AppSync endpoint revealed the Lambda null-pointer failure; the tool was removed from registration.

AI search rate limiting required upfront design for conversational patterns

ai_search

5 requests / minute

Protects conversational follow-up bursts.

request_email_discovery

10 requests / hour

Protects write-side expensive operations.

Rate limits are backend protection and LLM-loop protection.

Useful error responses are not the same as operational telemetry

What exists

- auth mode at startup
- mutation flag state
- tool count
- typed GraphQL errors
- explicit rate-limit errors

What should be built

- per-tool request logs
- latency
- input shape
- outcome
- error type

Start by assuming the MCP layer is a first-class interface

1. Design narrow tools
2. Separate read and write behavior early
3. Resist optimizing first for flexibility
4. Block mutations by default
5. Keep auth inside the server boundary
6. Validate with mocks, then the real backend
7. Treat logging and auditability as baseline requirements

Make the interface narrow, observable, testable, and safe enough for real workflows

- Start with one narrow read tool
- Add one default-deny mutation
- Test the actual request shape
- Validate the real backend before connecting the LLM client

Shadi Elyafi