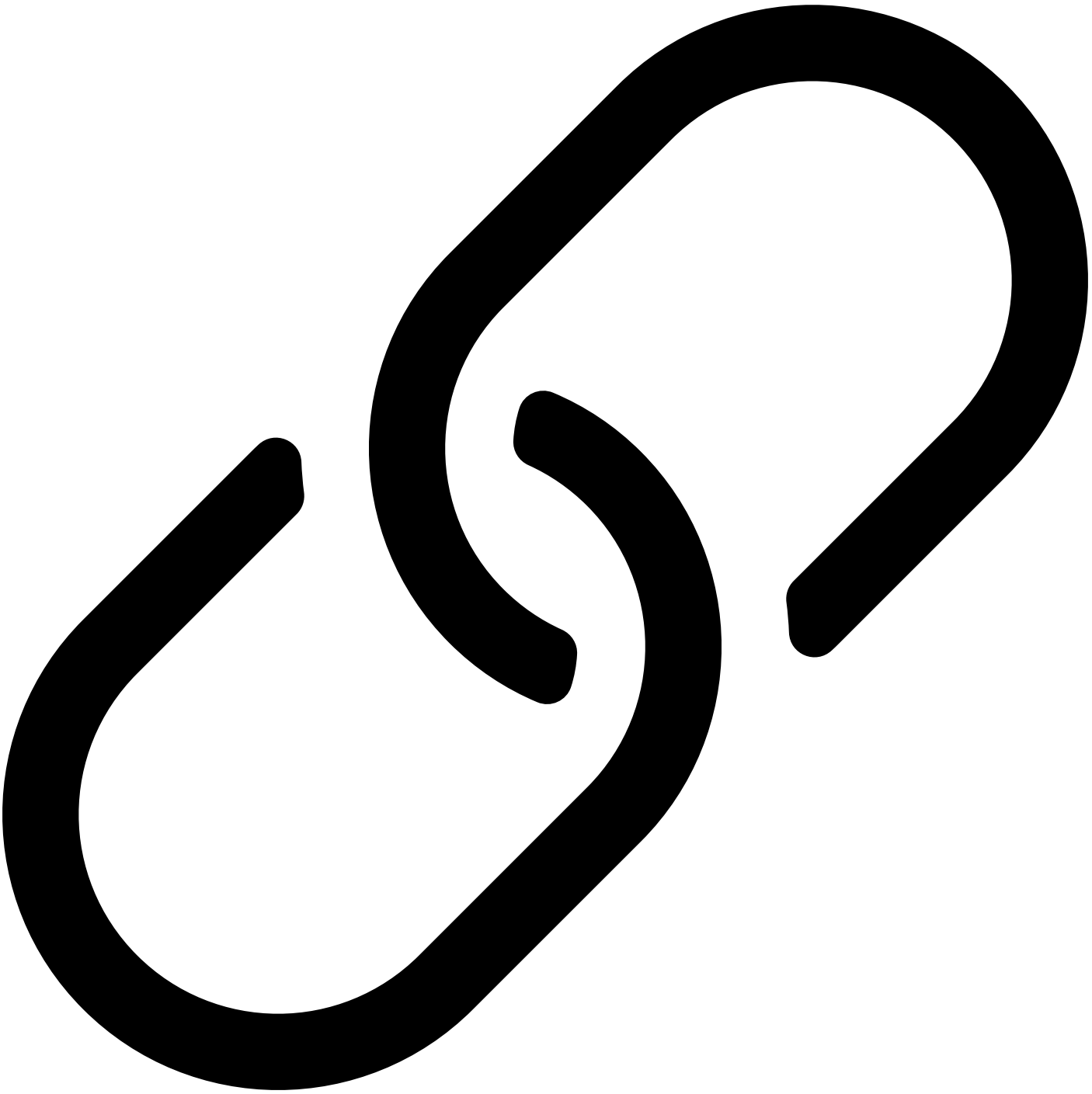
Building a Production-Ready Go Client Library: Considerations

Nirdosh Gautam

A Client library(SDK) is used as a tool to interact with the external services and its response in a structured format which is easy to integrate into existing software.

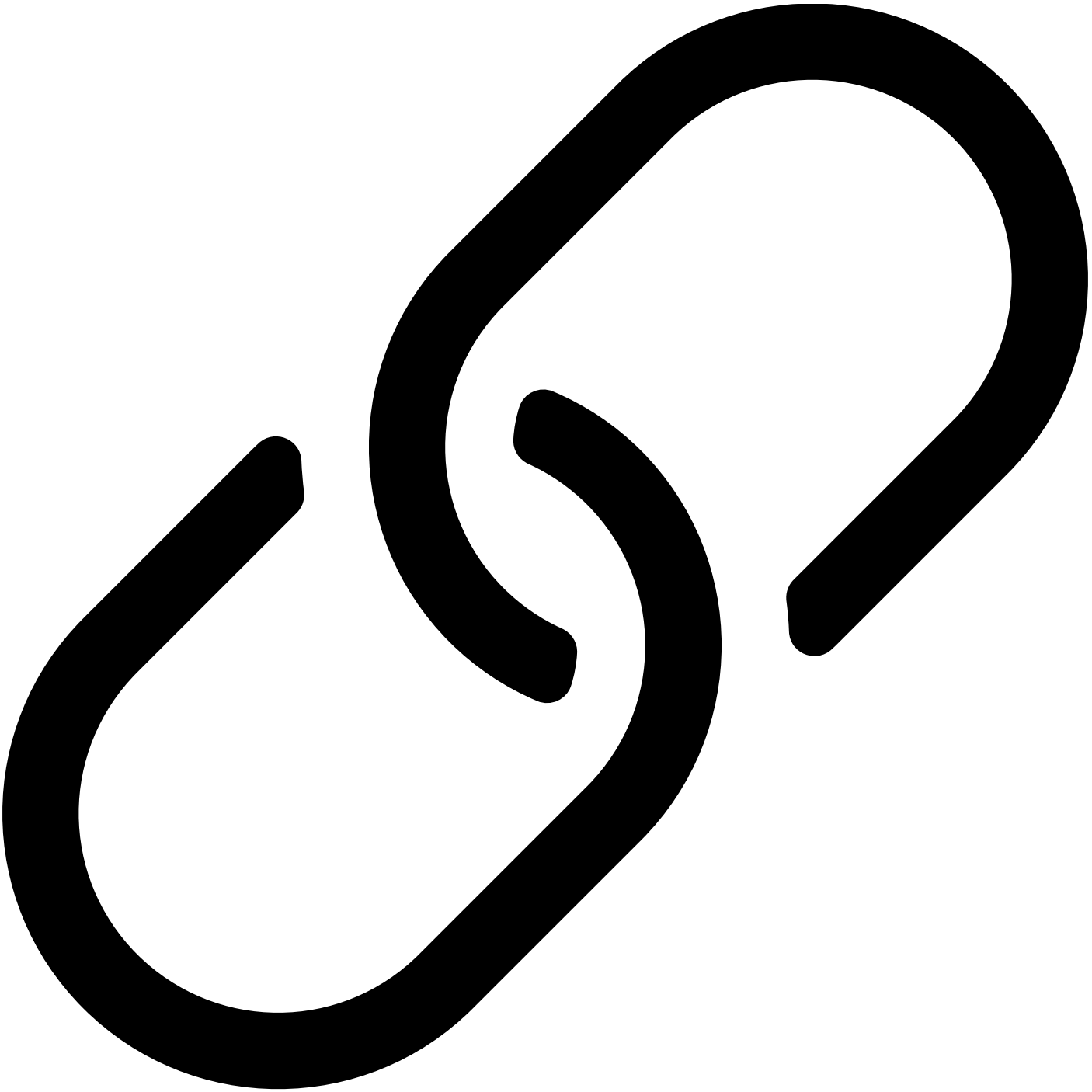
In this post, I will list some important features to consider while building an SDK in general and also talk about Golang-specific things.



1. Authentication

Authentication is done by the server. The SDK receives credentials as input and embedds the credentials while making requests to the server.

The library is an open-source code(most of the time), so we don't include any sensitive information like keys while publishing the public library.

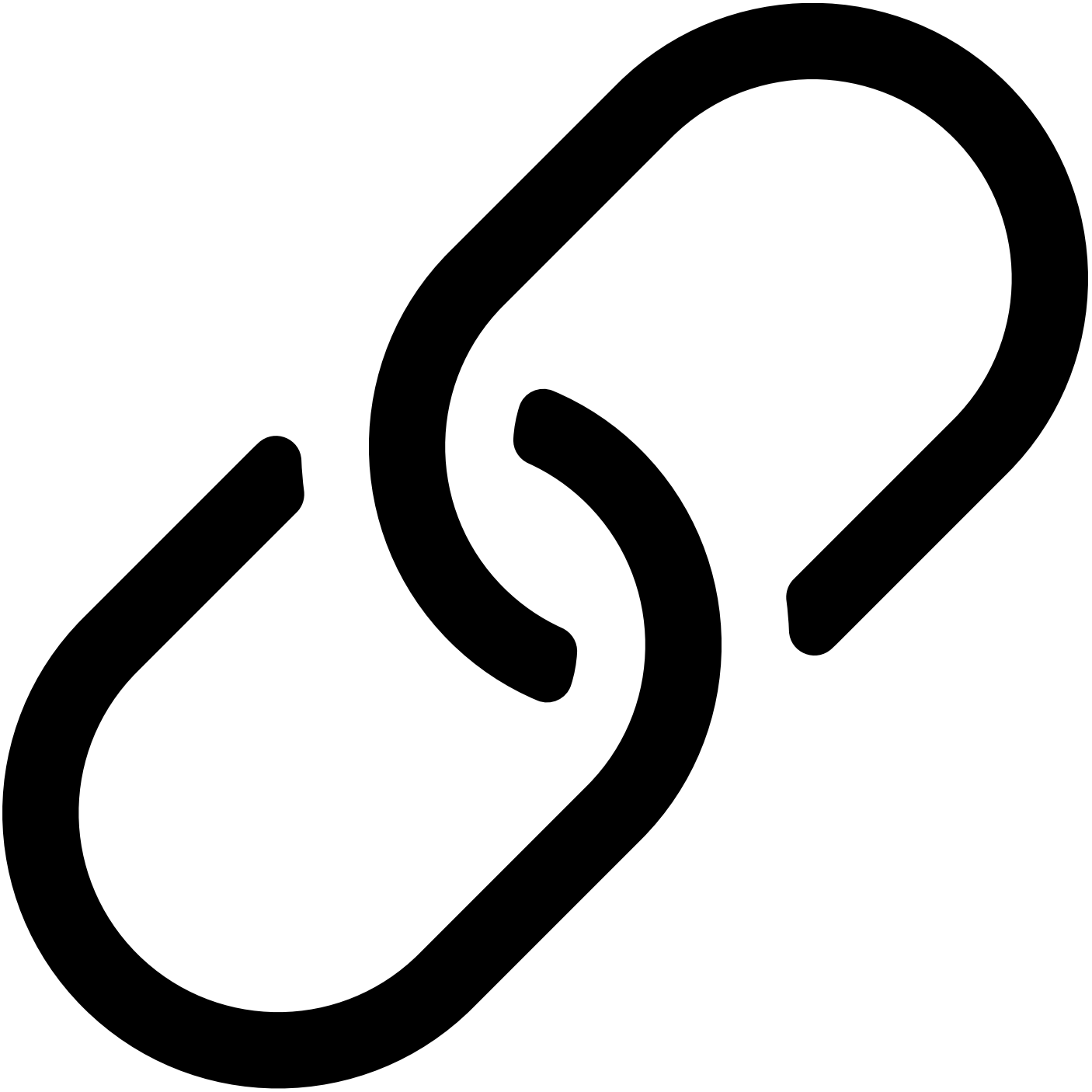


2. Input Validation

User input which does not depend on any extra information from the server can be validated locally in the SDK itself to avoid unnecessary load to the server. e.g. input types, size limits, etc.

Other inputs that require further information to validate should be delegated to the server. If user input is invalid, then the SDK can parse the 4XX response from the server and return a proper error message from

the SDK.



3. Configuration

The most common configurations in the Go library are:

Auth options

Able to use Custom HTTP Client

Able to pass context for custom cancellations and timeouts

Able to override timeouts, retries, base API endpoints, etc

How configuration should work:

Client library should allow easy initialization of library with default options which works without any

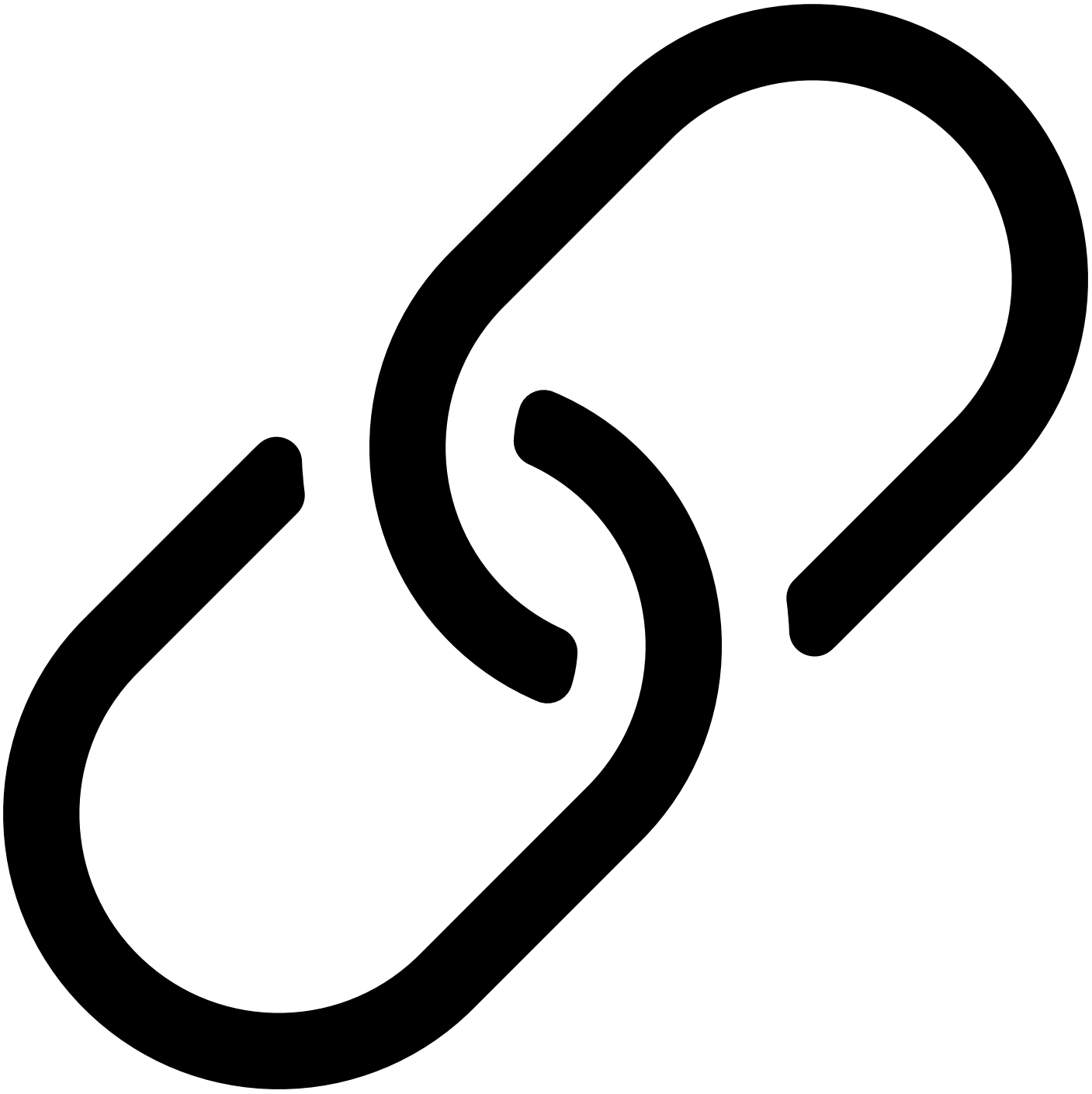
further setup.

It should allow configurations to be overridden easily with clear steps on how to do it.

Should focus on giving visibility of options to the developers. So, public interfaces and methods should have proper comments in the format supported by Godoc.

Configurator patterns: Method chaining and functional option patterns are easy to use and are widely adopted.

```
// method chaining
api.NewConfig().WithHTTPClient(client).WithMaxRetries(3)
```



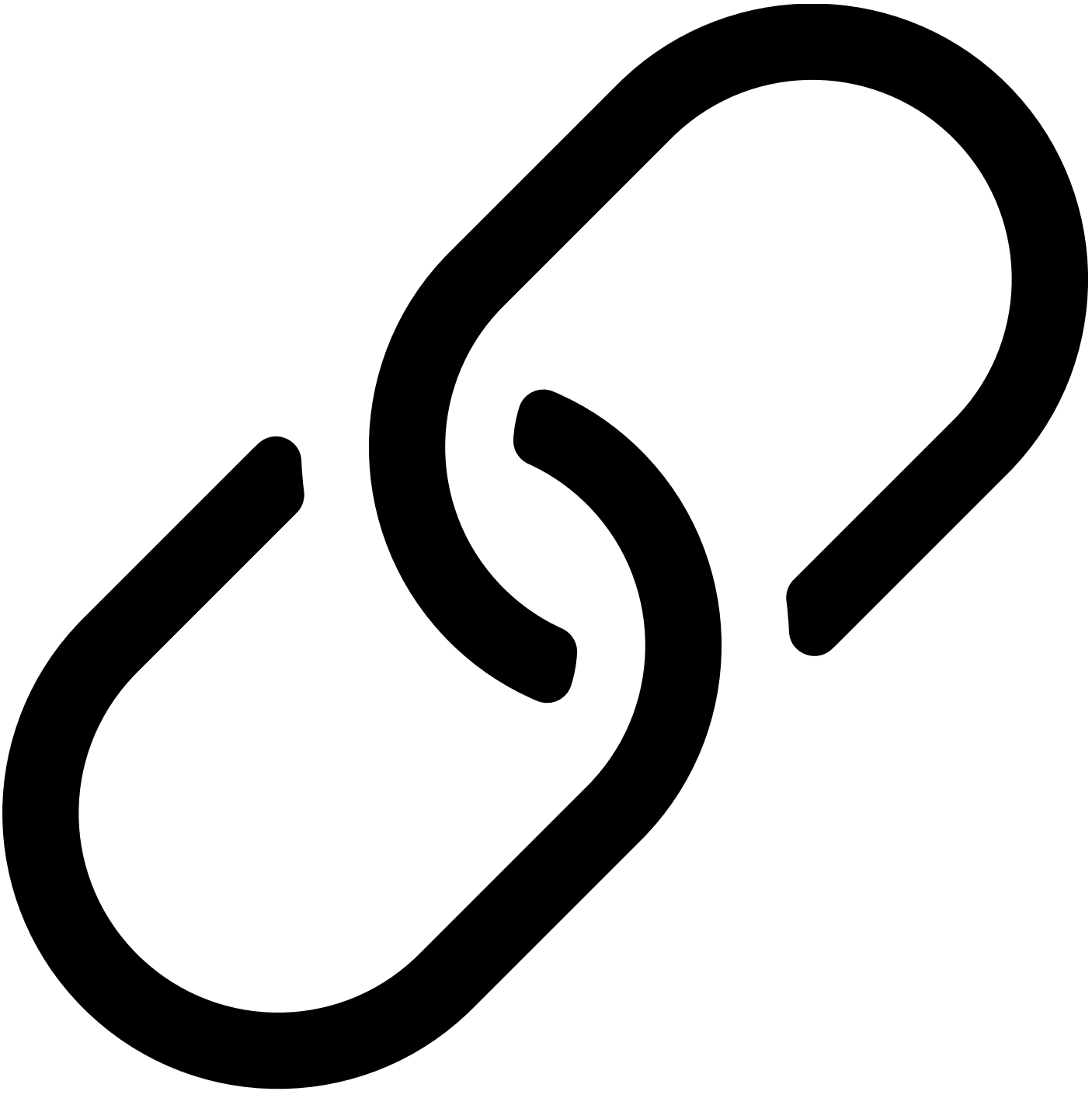
3. Error handling

Instead of returning a plain string as an error from the SDK, it is more useful if we return a ***structured error type*** by implementing Go's error interface.

Why?

Consumers may want to implement a custom logic(e.g. retry mechanism) based on the error(e.g. 429 Rate Exceeded) and searching for a word in the error string to categorize the error is certainly not reliable.
So, If we have custom error types, we can do type assertions to identify and categorize errors reliably.

```
type APIError struct {
    StatusCode int     // server response code
    Type       string  // err code e.g. TOO_MANY_REQUESTS
    Message    error   // full error
}
```



4. Retry mechanism

It is important to have some kind of retry logic in the client library so that transient server issues do not affect the consumers.

Some common retry algorithms:

Fixed window-based retry - *e.g. retry 5 times with a gap of 2 seconds*

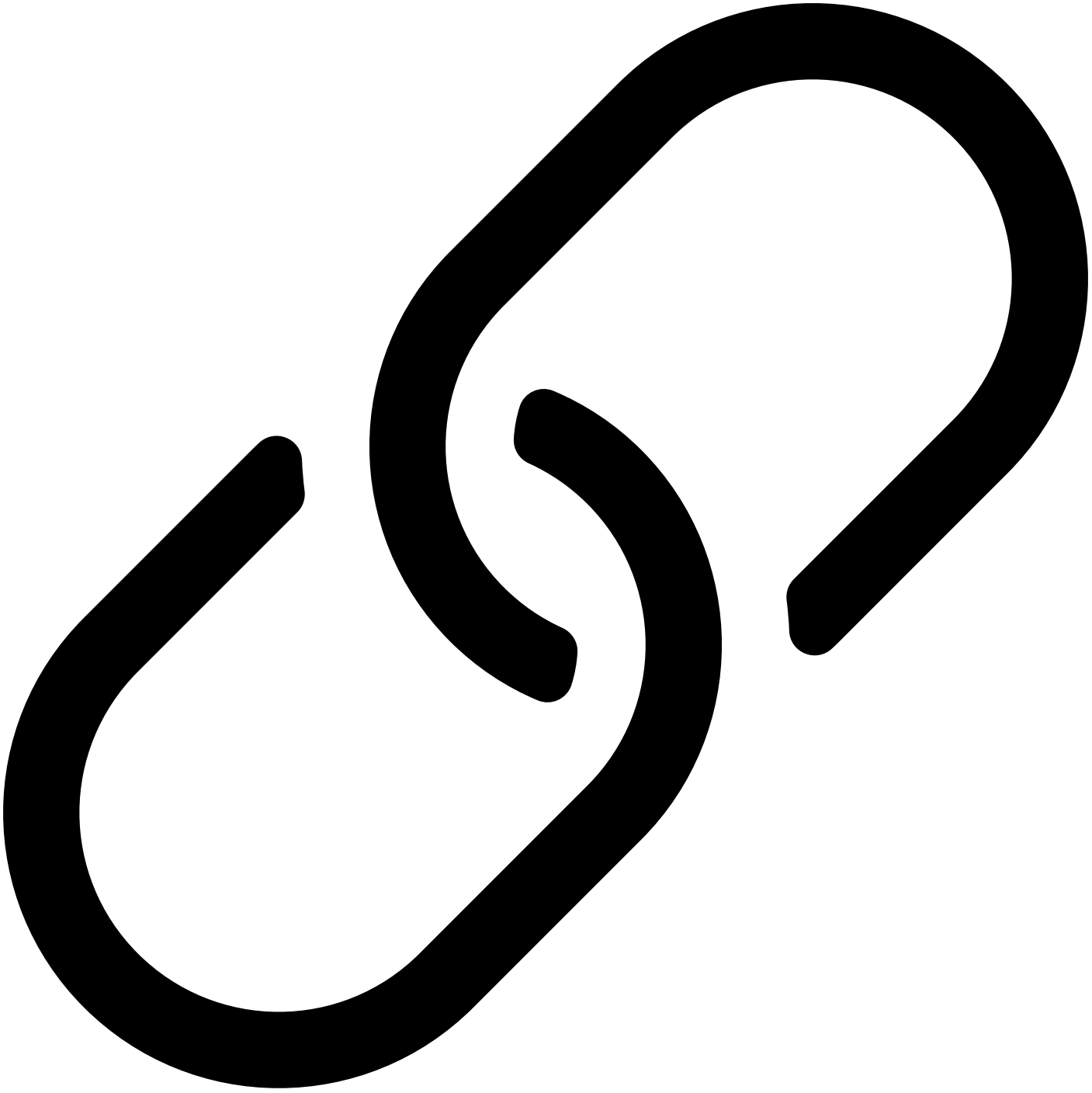
Exponential backoff - exponentially increases the delay between retries

Exponential backoff with jitter - have some randomness in exponential backoff

We can also give the option to use a custom retry function by exposing an interface. It is used in the case where we want to implement a custom retry behavior that is not offered by the library. For example, linear + exponential backoff with jitter

```
type Retryer interface {
    // retryer takes a function as argument and runs it N times
    // you are free to choose how Run function will implement retry
behavior
    Run(ctx context.Context, fn func(ctx context.Context) error) error
}
```

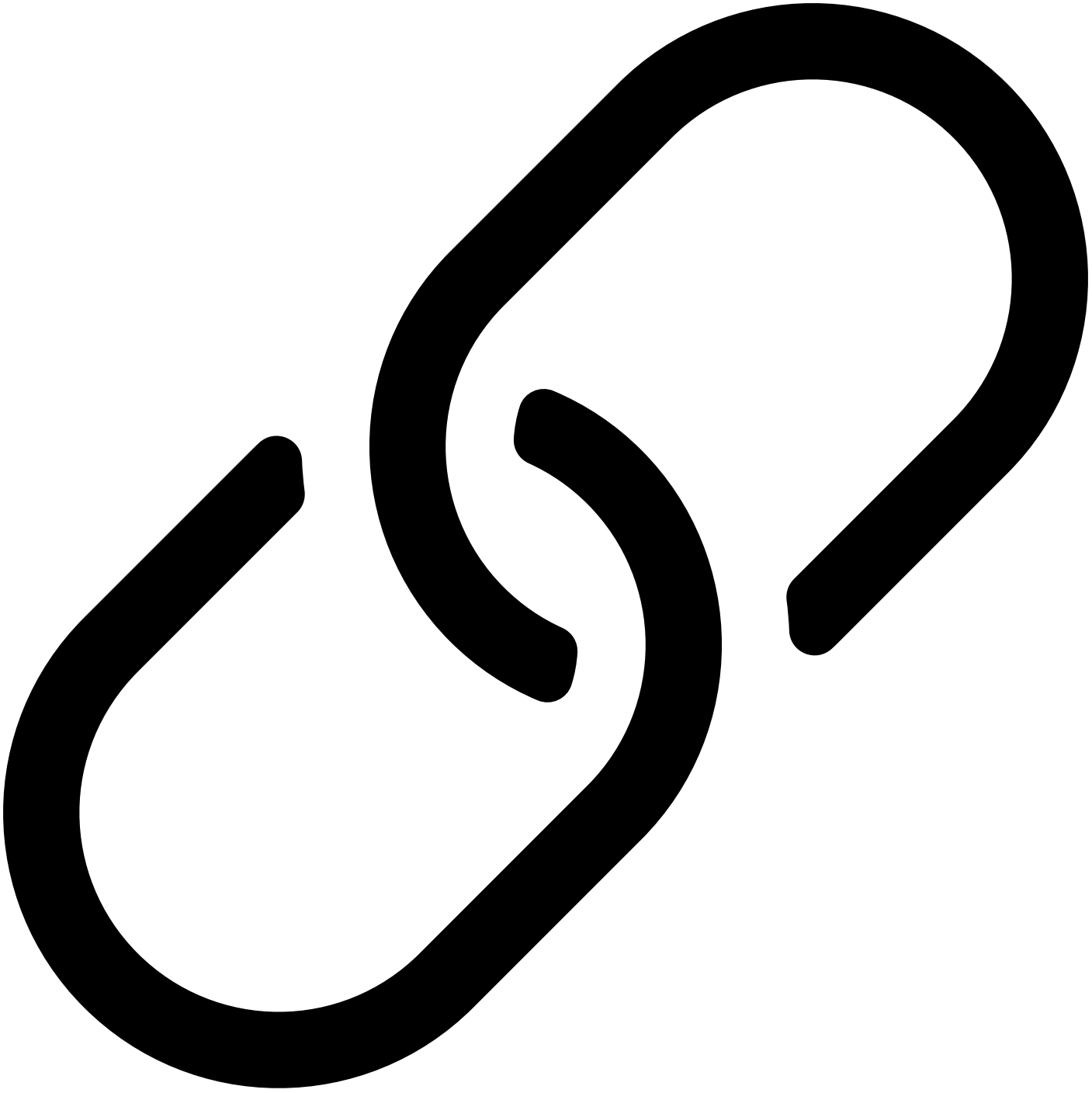
```
func testFunction(ctx context.Context) (string, error) {
    return "", errors.New("err from fn")
}
var fnResp string
var err error

retryer := &Retry{Delay: 2 * time.Second, MaxRetries: 3}
err := retryer.Run(ctx, func(ctx context.Context) error {
    fnResp, err = testFunction(ctx)
    return err
})
```



5. Dependencies

Avoid using external packages. If needed, create a common library to use inside your company.

If external dependencies are unavoidable, manage them using Go modules.

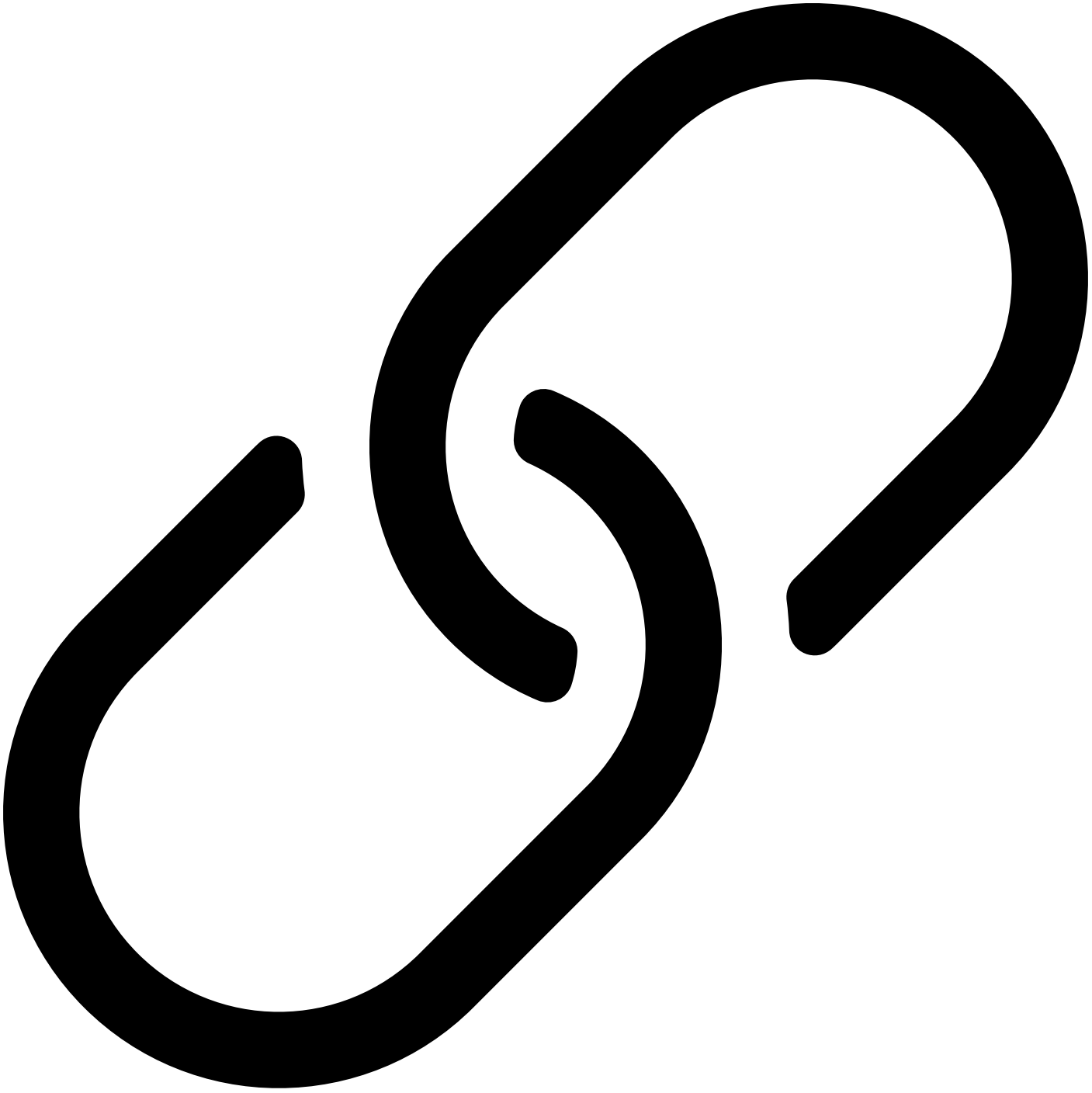


6. Logging

Log important events

Allow verbose logging where applicable e.g. printing out HTTP request/response objects

We can create a custom logging interface to allow developers to use their own logging implementation.

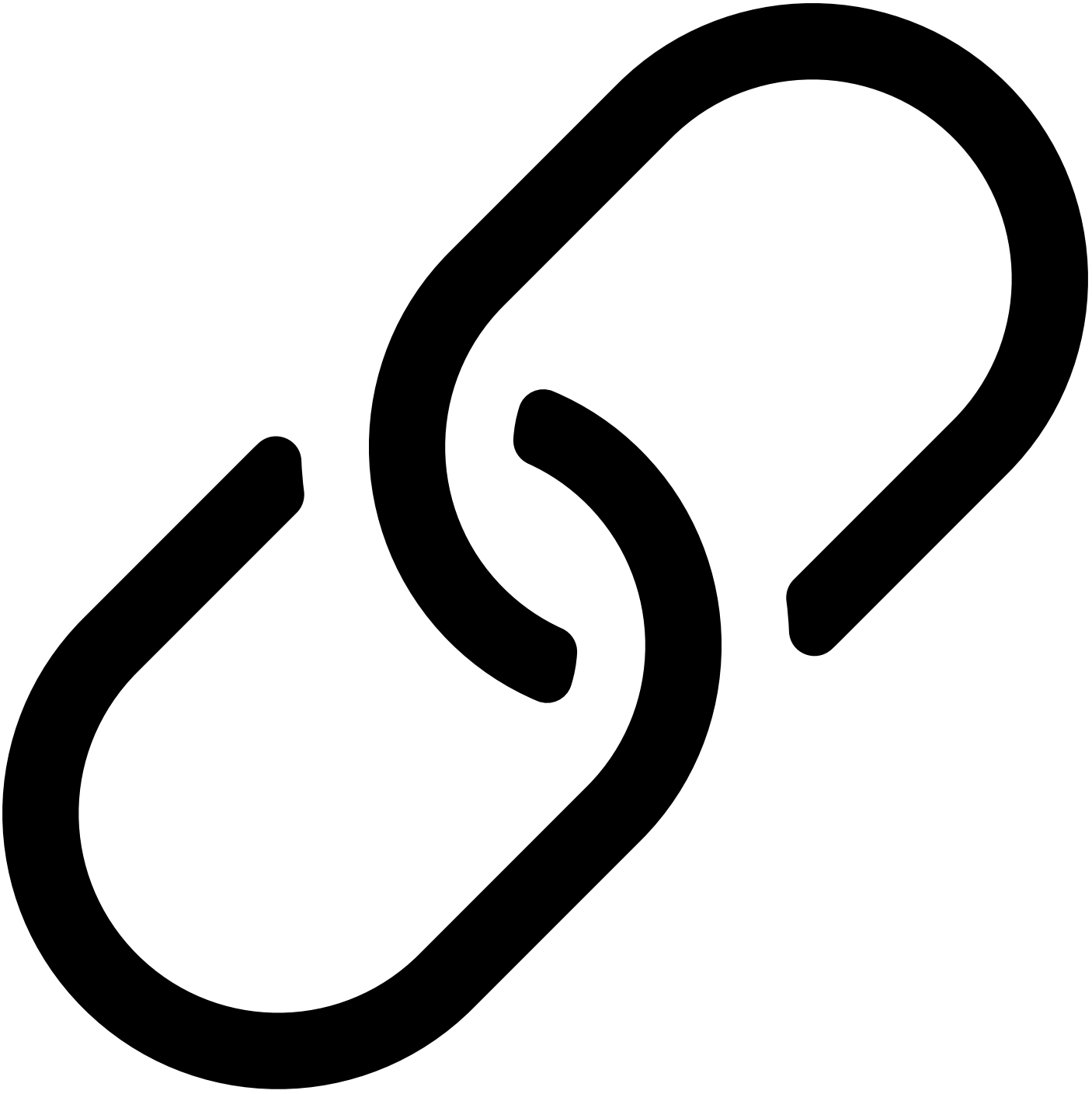


7. Tests

Have a good test coverage

Be aware that auto-generated SDK may not cover all scenarios.

Add extensive tests including all scenarios, esp for unhappy paths.



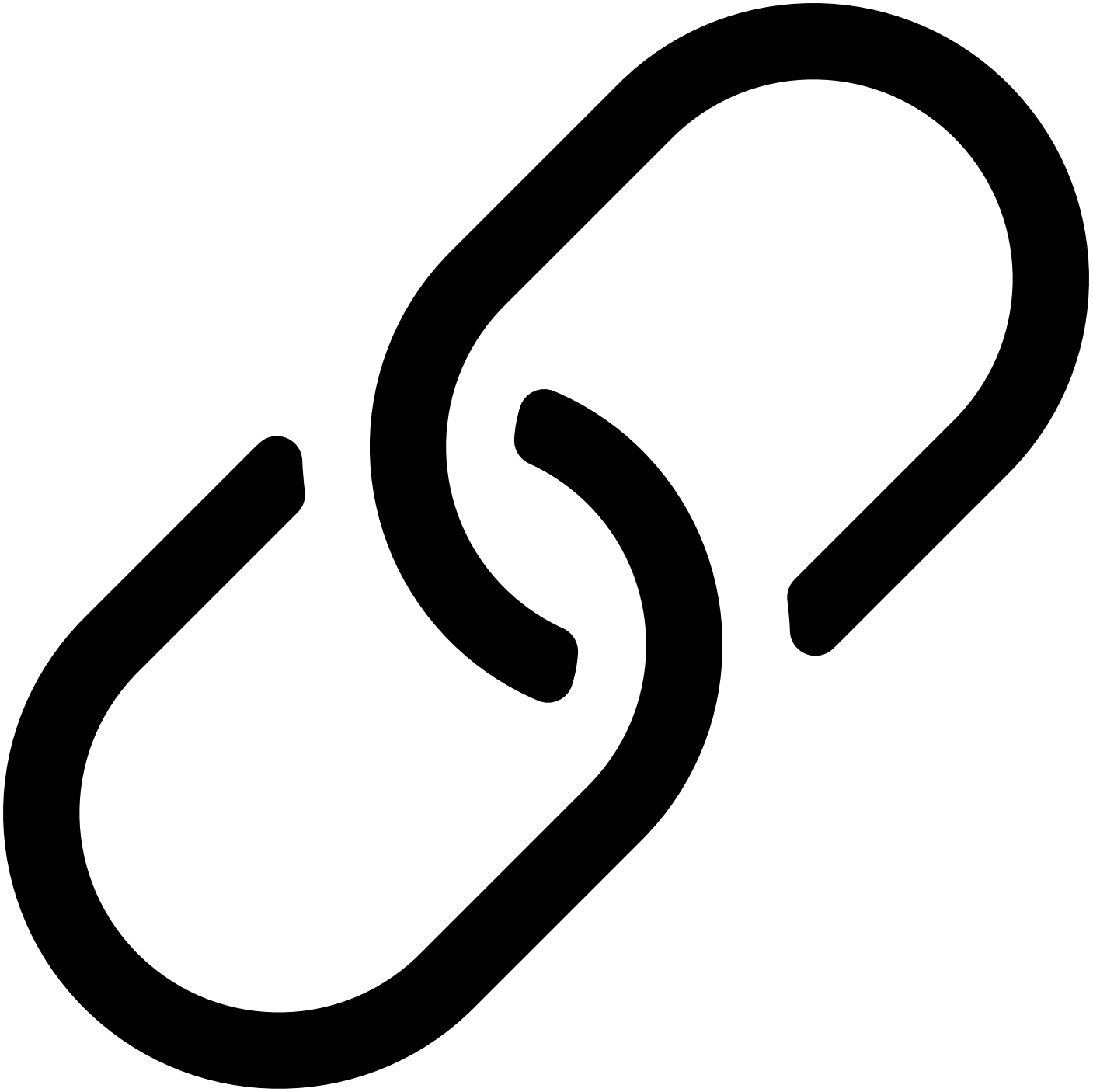
8. Documentation

Keep in mind that the package will be used by humans. So, you should focus on what information to share and how to make it easily understandable.

README should provide details regarding how to install and use the package with proper examples.

Add comments for public packages, variables, interfaces, methods, and structs.

Only export necessary types and interfaces to minimize noise in documentation(pkg.go.dev) and auto-suggestion in code editors like VSCode.



9. Publishing and Versioning

Go libraries are versioned using semantic versioning: `vMajor.Minor.Patch`

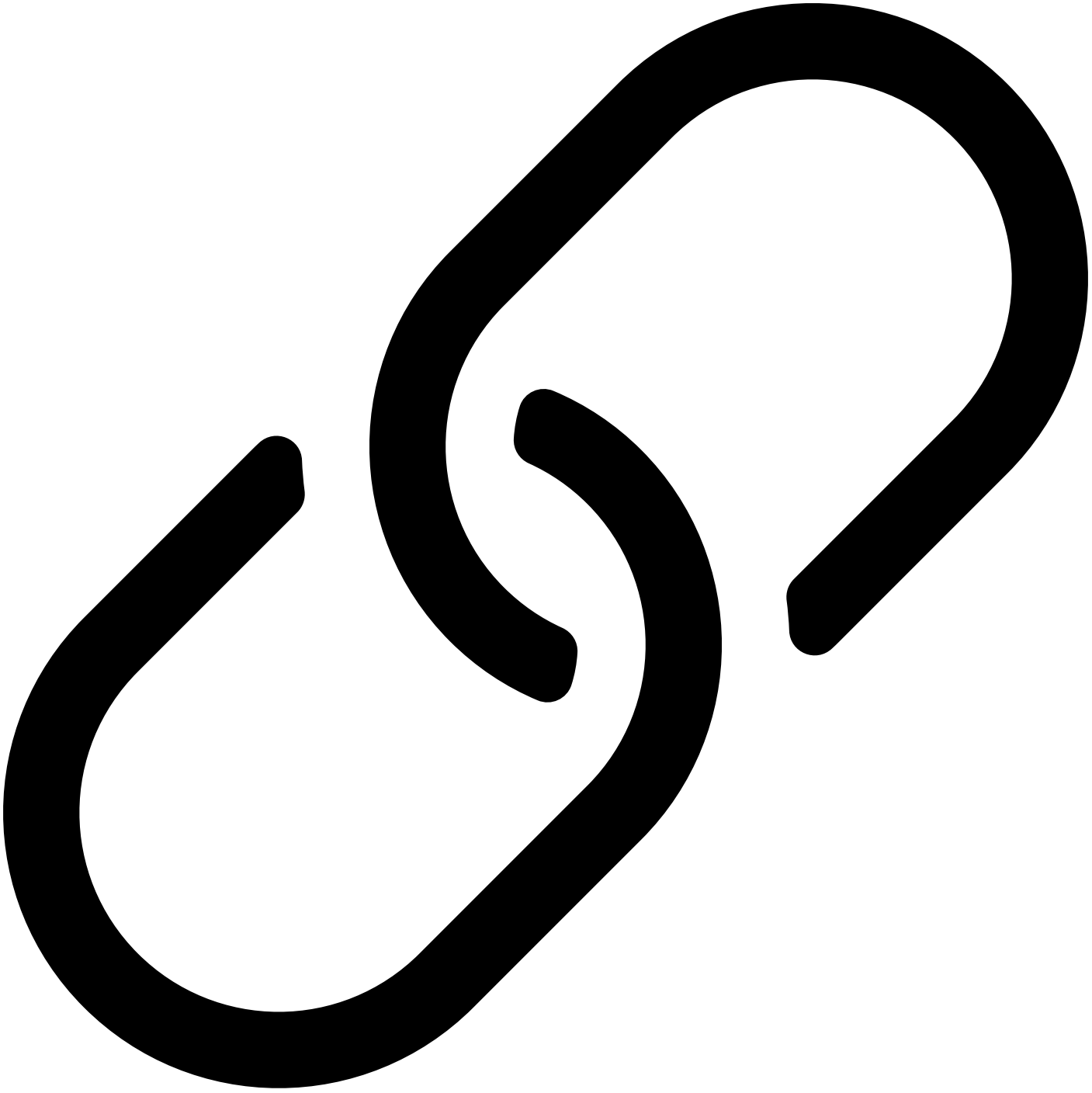
Make the library `go get` compatible.

Publish library with tags e.g. `v1.0.0`.

If a single library contains multiple APIs(e.g. aws-sdk-go), scope services in subfolders so that it is easy for the consumers to use.

e.g. `/services/s3`, `/services/ec2`

Tag changes of each API(e.g. s3, ec3) separately, so that different APIs will independently update their package versions at the same point in time.



10. Code Generation

Maintaining SDKs manually and keeping them up to date with new changes requires considerable time and resources.

Documenting our APIs in specs like Open API, allows us to generate client libraries in different languages including Go. Having said that, there will be cases where we have to add custom logic on top of the generated code.

There are tools like OpenAI Generator which take API specs via Swagger or open API spec files and then generate SDK for you in many languages.

```
openapi-generator generate -i swagger.yaml -g go
```

It's a good starting point for building SDKs.

Example Go SDK

If you want to have a look at the above features implemented in a sample SDK, check this repo:

github.com/nirdosh17/go-sdk-template