How to Rewrite a Service

19 April 2023

Michal Bock
Software Engineer, Deliveroo

Why rewrite a service?

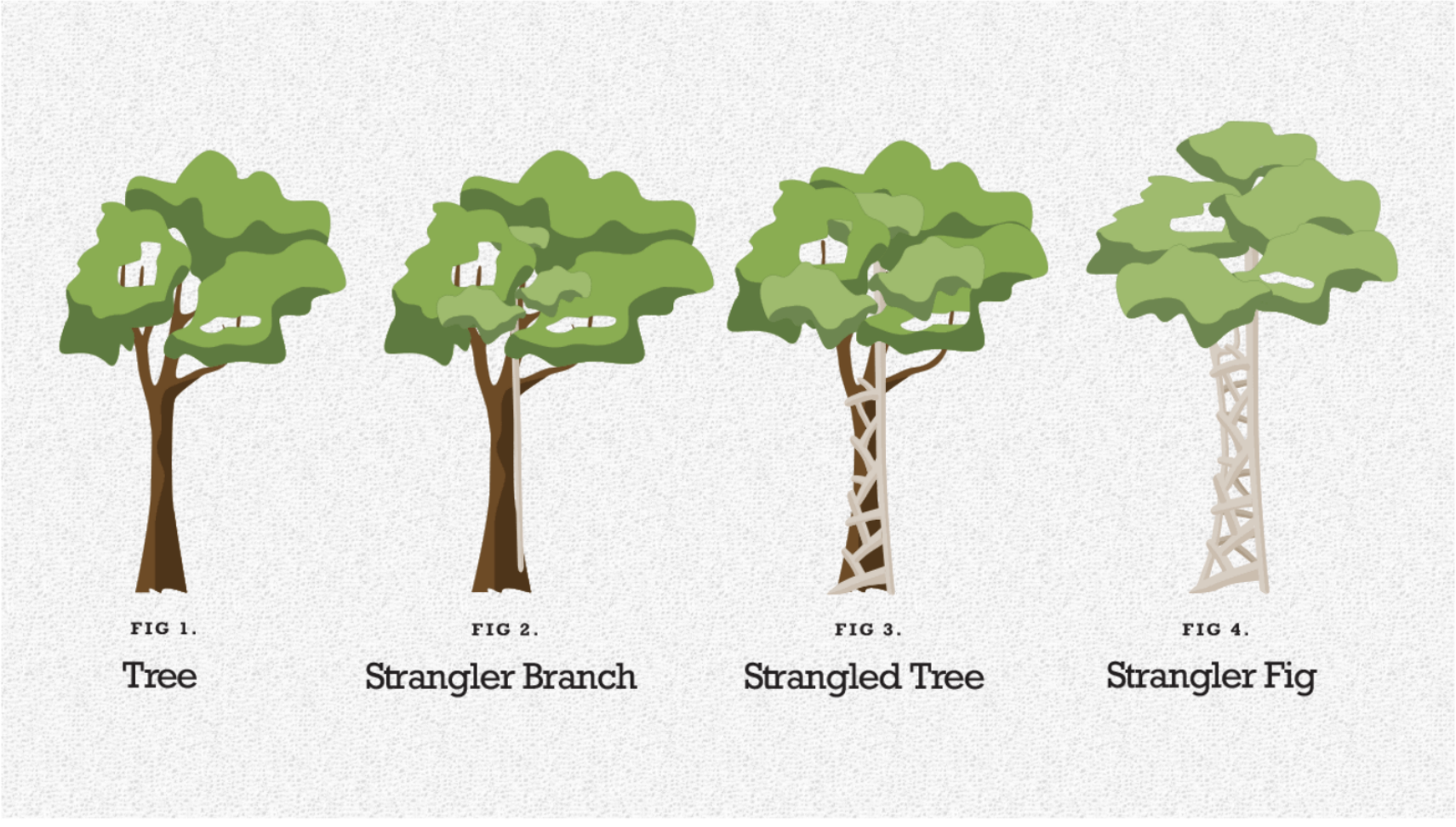
Convert it to another language

- Improve performance and resource usage
- Gain compatibility with newer technologies
- Aligning with the rest of the stack and engineer experience

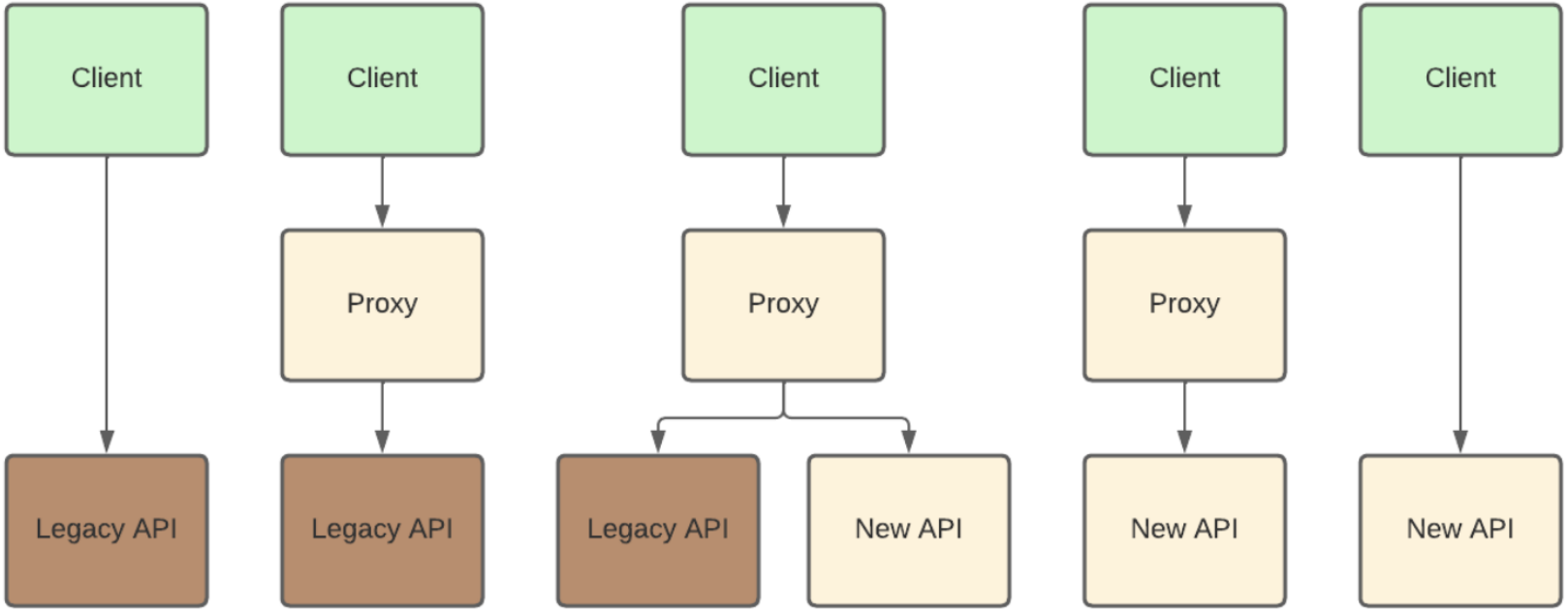
Goals for the Process

- Minimise the risk of causing an incident
- Ensure changes can be rolled out incrementally
- Ensure the old and new API behave the same way

The Strangler Fig Pattern



The Strangler Fig Pattern



patterns/strangler-fig

Let's Look at a Concrete Example

Assumptions

- We are replacing a REST api
- Both old and new API use the same database
- Proxy and the new API are written in Go

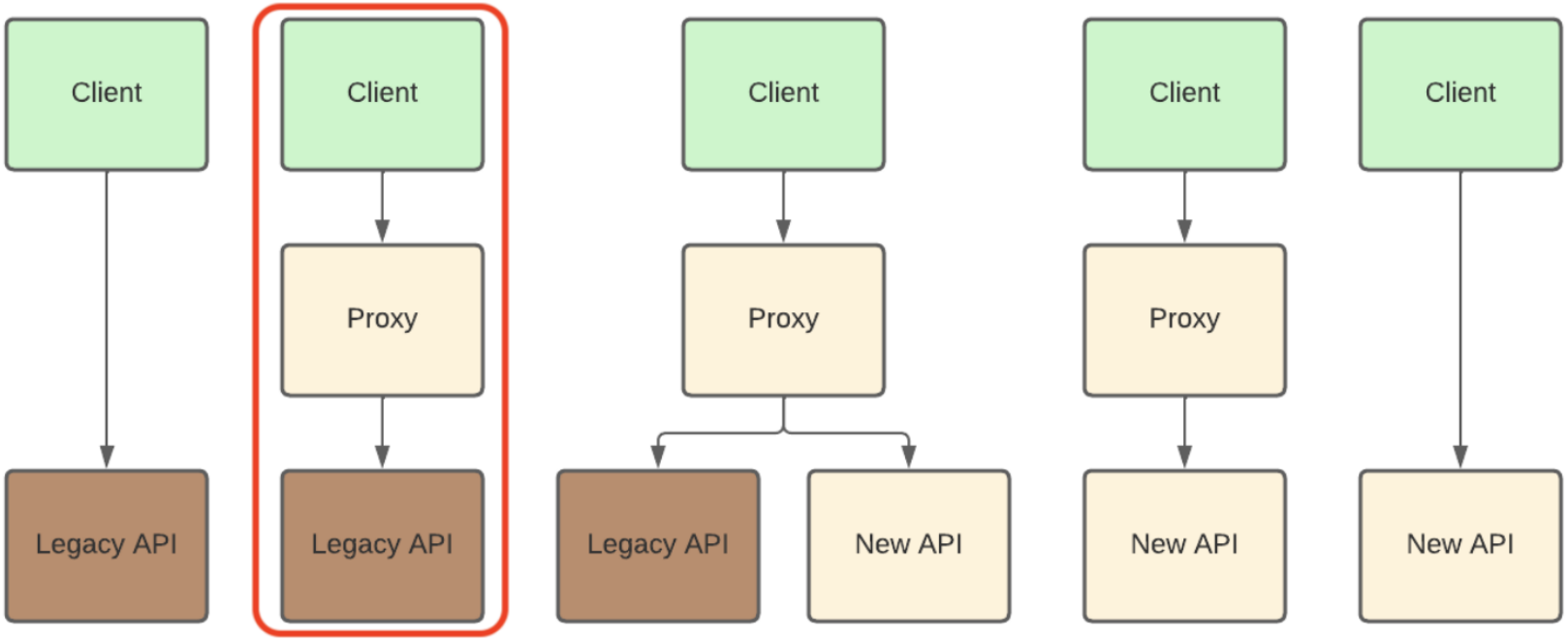
Building the new API

- This a common task that we won't address here
- We need to be careful with data writes during the transition to avoid double writes or data race

Rollout

1. Introduce HTTP proxy in front of the Old API
2. Route requests to both New and Old API
    - Serving results from the Old API
    - Alert on differences between the responses from the old and new API
3. Fix differences and repeat
4. Once we get 100% match we are ready to deploy the new version

Let's Build a Simple Proxy



HTTP Types from the Standard Library

```
type Handler interface {
    ServeHTTP(w ResponseWriter, req *Request)
}

type ResponseWriter interface {
    Header() Header
    Write([]byte) (int, error)
    WriteHeader(statusCode int)
}

type Request struct {
    Method string
    URL    *url.URL
    Header Header
    Body   io.ReadCloser
    //...
}

type Header map[string][]string
```

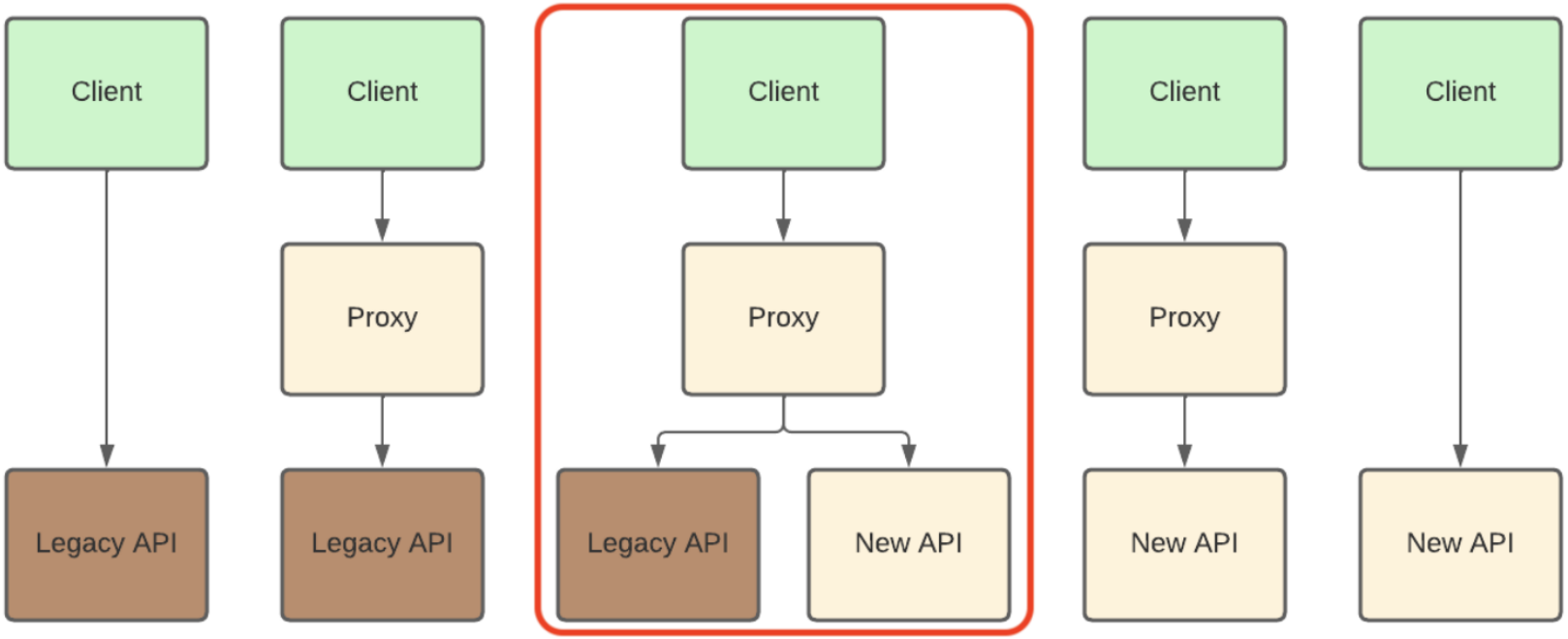
Let's Build a Simple Proxy

```
package main

import (
    "net/http"
    "net/http/httputil"
    "net/url"
    "os"
)

func run() error {
    oldServiceURL, err := url.Parse(os.Getenv("OLD_SERVICE_URL"))
    if err != nil {
        return err
    }
    server := http.Server{
        Addr:    ":3000",
        Handler: httputil.NewSingleHostReverseProxy(oldServiceURL),
    }
    return server.ListenAndServe()
}
```

Support for Proxying to Two Services



Support for Proxying to Two Services

```
type Manager interface {
    UseOld(r *http.Request) bool
}

func newProxyHandler(manager Manager, oldSvcURL, newSvcURL *url.URL) http.Handler {
    oldServiceHandler := httputil.NewSingleHostReverseProxy(oldSvcURL)
    newServiceHandler := httputil.NewSingleHostReverseProxy(newSvcURL)

    return http.HandlerFunc(func(w http.ResponseWriter, req *http.Request) {
        if manager.UseOld(req) {
            oldServiceHandler.ServeHTTP(w, req)
        } else {
            newServiceHandler.ServeHTTP(w, req)
        }
    })
}
```

Support for Checking for Differences between Responses

```
type Manager interface {
    GetProxyMode(r *http.Request) ProxyMode
}

type ProxyMode int

const (
    ProxyModeUseOld ProxyMode = iota
    ProxyModeUseNew
    ProxyModeUseOldAndDiff
)
```

Support for Checking for Differences between Responses

```
func newProxyHandler(manager Manager, oldSvcURL, newSvcURL *url.URL) http.Handler {
    oldServiceHandler := httputil.NewSingleHostReverseProxy(oldSvcURL)
    newServiceHandler := httputil.NewSingleHostReverseProxy(newSvcURL)
    diffHandler := newDiffHandler(oldServiceHandler, newServiceHandler)

    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        switch manager.GetProxyMode(r) {
        case ProxyModeUseOld:
            oldServiceHandler.ServeHTTP(w, r)
        case ProxyModeUseNew:
            newServiceHandler.ServeHTTP(w, r)
        case ProxyModeUseOldAndDiff:
            diffHandler.ServeHTTP(w, r)
        }
    })
}
```

The Diffing Handler

```
func newDiffHandler(oldHandler, newHandler http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, req *http.Request) {
        oldHandlerReq, newHandlerReq, err := getRequestsToForward(req)
        if err != nil {
            http.Error(w, "failed to read payload", http.StatusInternalServerError)
            return
        }
        oldHandlerWriter, newHandlerWriter := httptest.NewRecorder(), httptest.NewRecorder()

        diffWG := &sync.WaitGroup{}
        diffWG.Add(2)
        // Asynchronously check for differences after both handlers are done.
        go diffResponses(diffWG, oldHandlerWriter, newHandlerWriter)

        go func() {
            defer diffWG.Done()
            newHandler.ServeHTTP(newHandlerWriter, newHandlerReq)
        }()

        defer diffWG.Done()
        oldHandler.ServeHTTP(oldHandlerWriter, oldHandlerReq)
        copyResponse(oldHandlerWriter, w)
    })
}
```

Capturing the Request

```
func getRequestsToForward(req *http.Request) (*http.Request, *http.Request, error) {
    payload, err := io.ReadAll(req.Body)
    if err != nil {
        return nil, nil, err
    }
    oldHandlerReq := req.Clone(req.Context())
    oldHandlerReq.Body = io.NopCloser(bytes.NewReader(payload))

    newHandlerReq := req.Clone(context.Background())
    newHandlerReq.Body = io.NopCloser(bytes.NewReader(payload))

    return oldHandlerReq, newHandlerReq, nil
}
```

Response Recorder

- The **net/http/httptest** package provides an implementation of the **http.ResponseWriter** interface that captures the response in form of the **ResponseRecorder** struct
- This is intended for capturing responses in tests, but we can use this to capture responses from both APIs
- We can then forward the response from the old API to the caller and afterwards compare the differences

The Diffing Handler

```
func newDiffHandler(oldHandler, newHandler http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, req *http.Request) {
        oldHandlerReq, newHandlerReq, err := getRequestsToForward(req)
        if err != nil {
            http.Error(w, "failed to read payload", http.StatusInternalServerError)
            return
        }
        oldHandlerWriter, newHandlerWriter := httptest.NewRecorder(), httptest.NewRecorder()

        diffWG := &sync.WaitGroup{}
        diffWG.Add(2)
        // Asynchronously check for differences after both handlers are done.
        go diffResponses(diffWG, oldHandlerWriter, newHandlerWriter)

        go func() {
            defer diffWG.Done()
            newHandler.ServeHTTP(newHandlerWriter, newHandlerReq)
        }()

        defer diffWG.Done()
        oldHandler.ServeHTTP(oldHandlerWriter, oldHandlerReq)
        copyResponse(oldHandlerWriter, w)
    })
}
```

Writing the Response to the Caller

```
func copyResponse(recorder *httptest.ResponseRecorder, w http.ResponseWriter) {
    for name, values := range recorder.Header() {
        for _, val := range values {
            w.Header().Add(name, val)
        }
    }
    w.WriteHeader(recorder.Code)
    _, _ = w.Write(recorder.Body.Bytes())
}
```

Checking for Differences

```
func diffResponses(wg *sync.WaitGroup, oldResponse, newResponse *httptest.ResponseRecorder) {
    wg.Wait() // Wait for both requests to finish.

    if oldResponse.Code != newResponse.Code {
        fmt.Printf("Status Code Diff Old: %v New: %v\n", oldResponse.Code, newResponse.Code)
    }
    if diff := cmp.Diff(oldResponse.Header(), newResponse.Header()); diff != "" {
        fmt.Println("Header Diff:", diff)
    }

    var oldJSON, newJSON any

    if err := json.Unmarshal(oldResponse.Body.Bytes(), &oldJSON); err != nil {
        fmt.Printf("failed to unmarshal old json: %s\n", err)
    }
    if err := json.Unmarshal(newResponse.Body.Bytes(), &newJSON); err != nil {
        fmt.Printf("failed to unmarshal new json: %s\n", err)
    }
    if diff := cmp.Diff(oldJSON, newJSON); diff != "" {
        fmt.Println("Body Diff:", diff)
    }
}
```

Checking for Differences

- We are using **github.com/google/go-cmp/cmp** to check for differences
  - It is intended to only be used in tests, so it's performance is not optimal
  - It may panic if it cannot compare the values
  - However, it is still usable in production with panic handling and generous CPU capacity
- Alternatively we can handwrite equal methods or generate them using **github.com/awalterschulze/goderive** or a similar tool

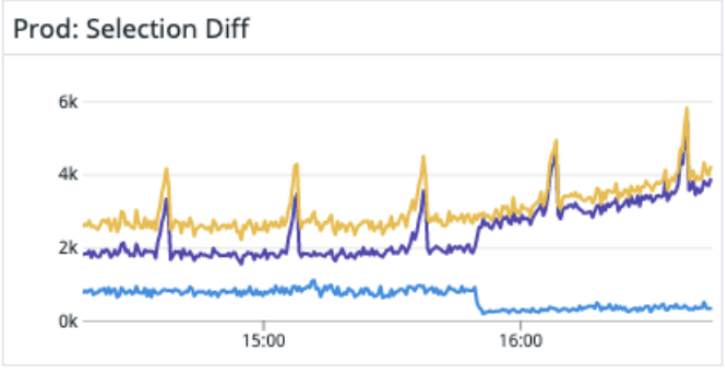
Production Setup

- Use metrics in addition to logs to track the number of matching responses and number of responses with differences
- Ensure tracing information is propagated
- Configure panic handling and improve other error handling
- Set timeout for requests going to the new API
- Control which API is used for which route via feature flags
- Use **GetBody** of **http.Request** to handle client retries correctly
- Write response to the client immediately instead of copying it once it's completely buffered.

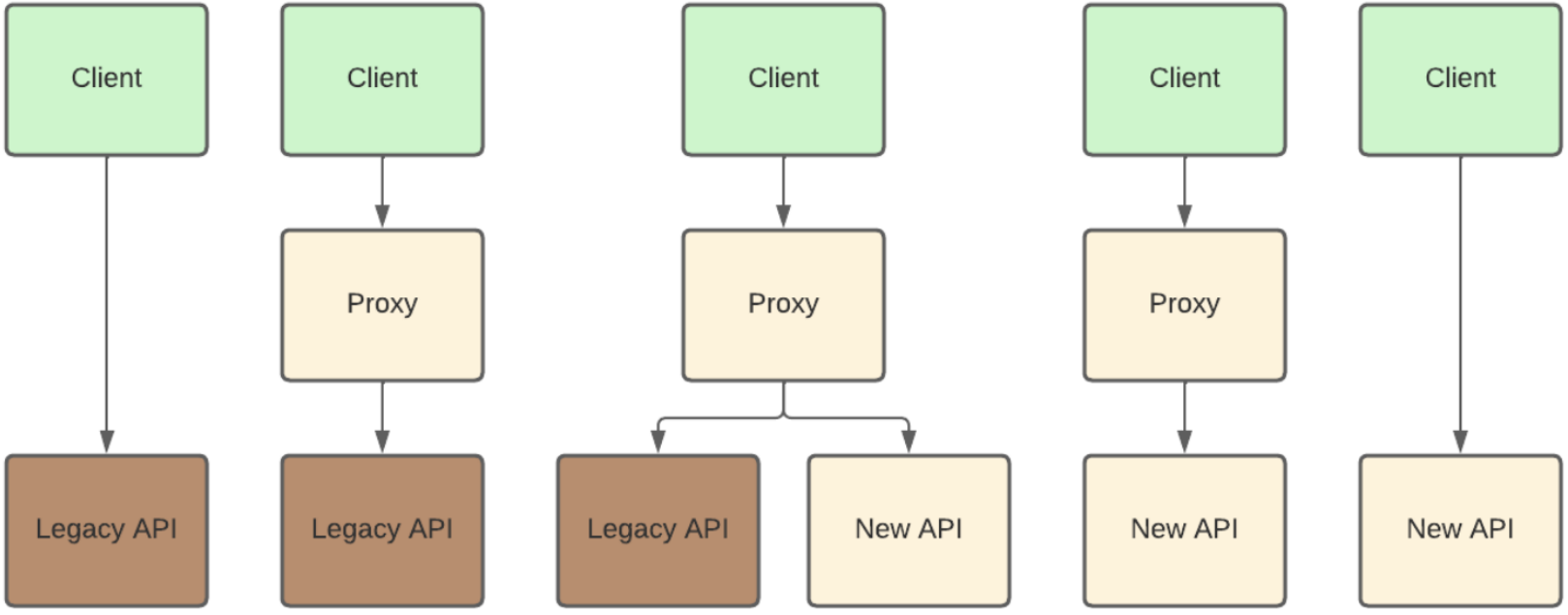
github.com/SpeedyCoder/http-strangler

How did this Approach Work for Us?

- Reached 100% match across all our read endpoints in production
- Completed the rewrite on time and didn't cause an incident
- Implemented the new api using gRPC and used the http proxy as a converter



The Strangler Fig Pattern



Thank you

Michal Bock
Software Engineer, Deliveroo
michal.bock@gmail.com
https://michalbock.com
@michal_bock