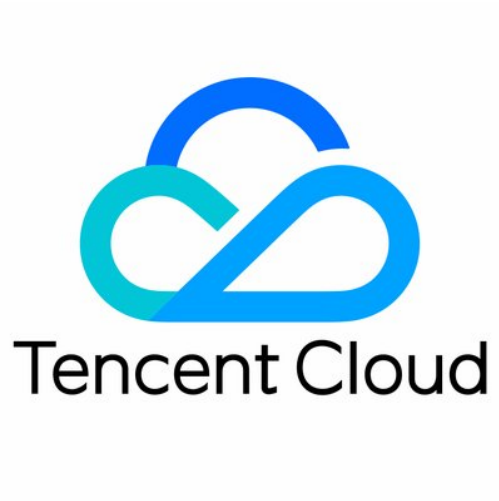
Application Performance Management Access Guide

Product Documentation



Contents

Access Guide
Accessing Go Application
Connecting Go Applications Using OpenTelemetry-Go (Recommended)

Last updated：2024-06-19 16:31:30

Note:
OpenTelemetry is a collection of tools, APIs, and SDKs for monitoring, generating, collecting, and exporting telemetry data (metrics, logs, and traces) to help users analyze the performance and behaviors of the software. For more information about OpenTelemetry, see the OpenTelemetry Official Website.
The OpenTelemetry community is active, with rapid technological changes, and widely compatible with mainstream programming languages, components, and frameworks, making its link-tracing capability highly popular for cloud-native microservices and container architectures.
This document will introduce how to connect Go applications with the community's OpenTelemetry-Go scheme. OpenTelemetry-Go provides a series of APIs so that users can send performance data to the observability platform's server. This document introduces how to connect Tencent Cloud APM based on OpenTelemetry Go through the most common application behaviors, such as HTTP services and database access. For more uses of OpenTelemetry-Go, see the Project Homepage.

Prerequisites

This scheme supports the officially supported versions of Go, currently 1.21 and 1.22. For lower versions, the connection is theoretically possible, but the community does not maintain full compatibility. For specific information, see the community's Compatibility Description.

Preliminary steps: Get the connect point and Token.

1. Log in to the TCOP console.
2. In the left menu column, select **Application Performance Management > Application monitoring**, and click **Application list > Access application**.
3. In the **Data access** drawer that pops up on the right, click the Go language.
4. On the **Access Go application** page, select the **Region** and **Business System** you want to connect.
5. Select **Access protocol type** as **OpenTelemetry**.

6. **Reporting method** Choose your desired reporting method, and obtain your **Access Point** and **Token**.

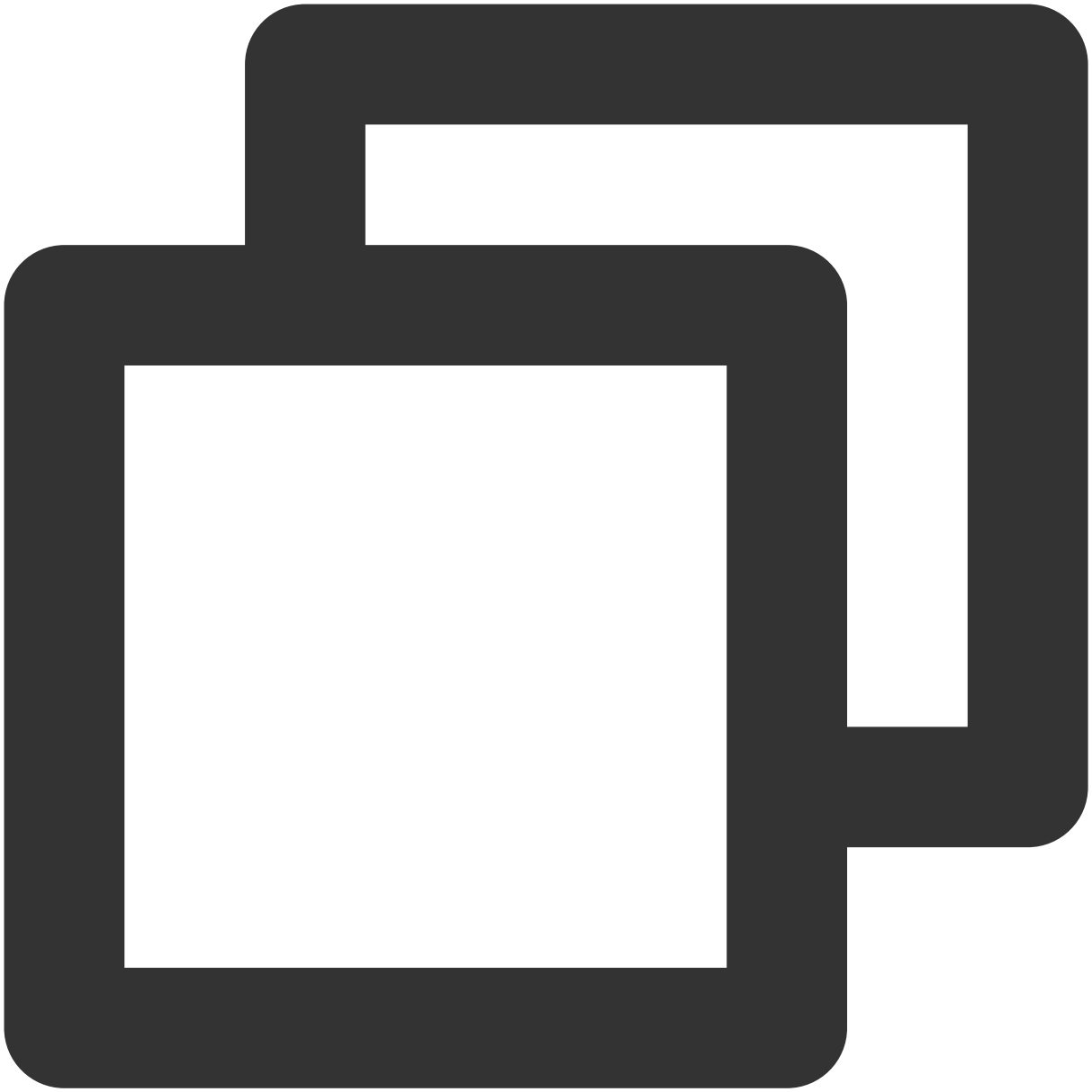
Note:

Private network reporting: Using this reporting method, your service needs to run in the Tencent Cloud VPC. Through VPC connecting directly, you can avoid the security risks of public network communication and save on reporting traffic overhead.

Public network reporting: If your service is deployed locally or in non-Tencent Cloud VPC, you can report data in this method. However, it involves security risks in public network communication and incurs reporting traffic fees.

Connecting Go Applications

Step 1: Introduce OpenTelemetry-related dependencies to implement the SDK initialization logic.



```
package main

import (
    "context"
    "errors"
    "go.opentelemetry.io/otel"
    "go.opentelemetry.io/otel/attribute"
    "go.opentelemetry.io/otel/exporters/otlp/otlptrace/otlptracegrpc"
    "go.opentelemetry.io/otel/propagation"
    "go.opentelemetry.io/otel/sdk/resource"
    "go.opentelemetry.io/otel/sdk/trace"
```

```
    sdktrace "go.opentelemetry.io/otel/sdk/trace"
    "log"
)

func setupOTelSDK(ctx context.Context) (*trace.TracerProvider, error) {
    opts := []otlptracegrpc.Option{
        otlptracegrpc.WithEndpoint("<endpoint>"), // Replace <endpoint> with the repo
        otlptracegrpc.WithInsecure(),
    }
    exporter, err := otlptracegrpc.New(ctx, opts...)
    if err != nil {
        log.Fatal(err)
    }
    r, err := resource.New(ctx, []resource.Option{
        resource.WithAttributes(
            attribute.KeyValue{Key: "token", Value: "<token>"}, // Replace <token> wit
            attribute.KeyValue{Key: "service.name", Value: "<servceName>"}, // Replace
            attribute.KeyValue{Key: "host.name", Value: "<hostName>"}, // Replace <hos
        ),
    }...)
    if err != nil {
        log.Fatal(err)
    }
    tp := sdktrace.NewTracerProvider(
        sdktrace.WithSampler(sdktrace.AlwaysSample()),
        sdktrace.WithBatcher(exporter),
        sdktrace.WithResource(r),
    )
    otel.SetTracerProvider(tp)
    otel.SetTextMapPropagator(propagation.NewCompositeTextMapPropagator(propagation.
    return tp, nil
}
```

The corresponding field descriptions are as follows, replace them according to actual conditions.

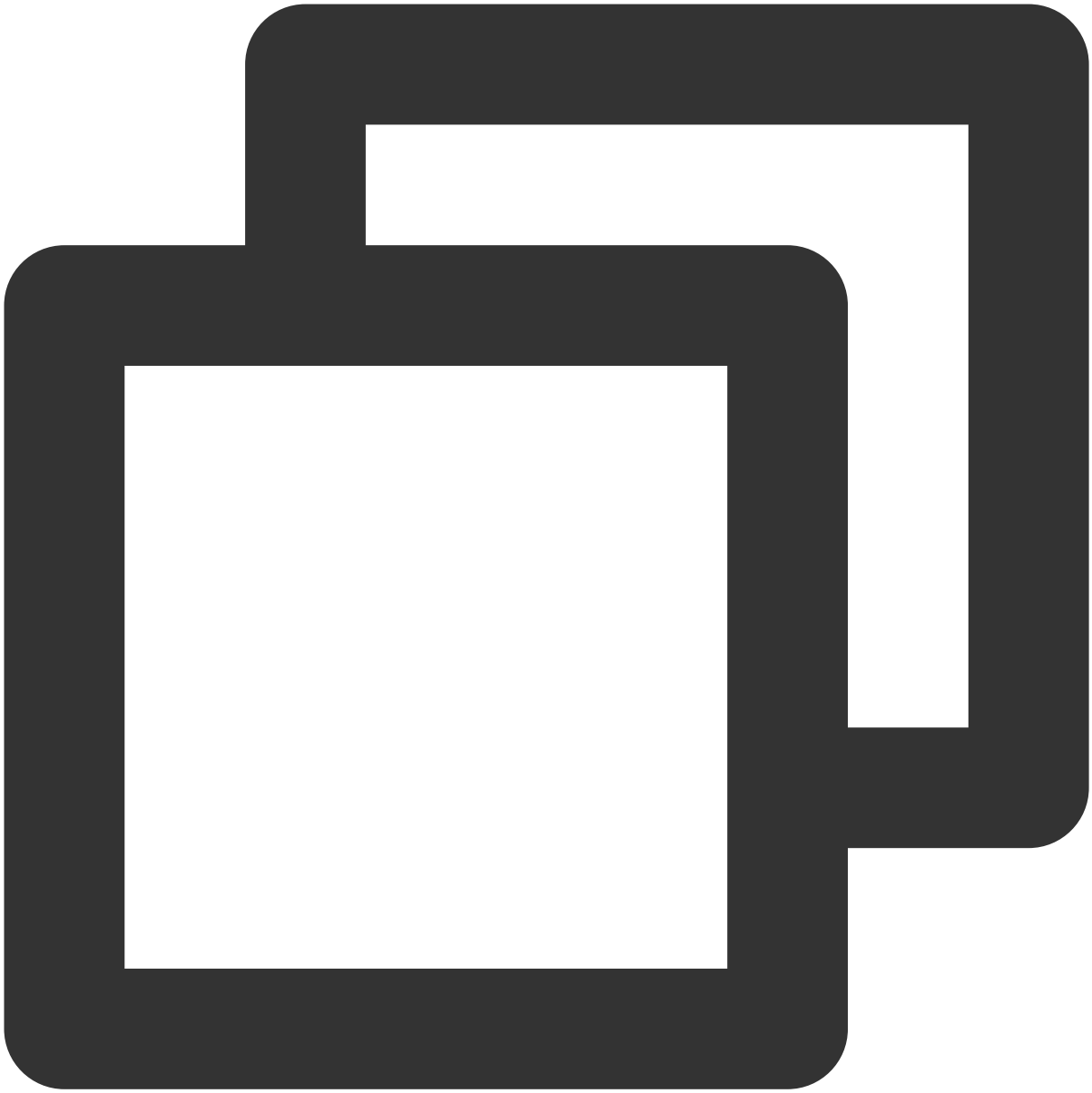
`<serviceName>` : Application name. Multiple application processes connecting with the same serviceName are displayed as multiple instances under the same application in APM. The application name can be up to 63 characters and can only contain lowercase letters, digits, and the separator (-), and it must start with a lowercase letter and end with a digit or lowercase letter.

`<token>` : The business system Token obtained in the preliminary steps.

`<hostName>` : The hostname of this instance, which is the unique identifier of the application instance. It can usually be set to the IP address of the application instance.

`<endpoint>` : The connect point obtained in the preliminary steps.

Step 2: SDK initialization and start the HTTP service.

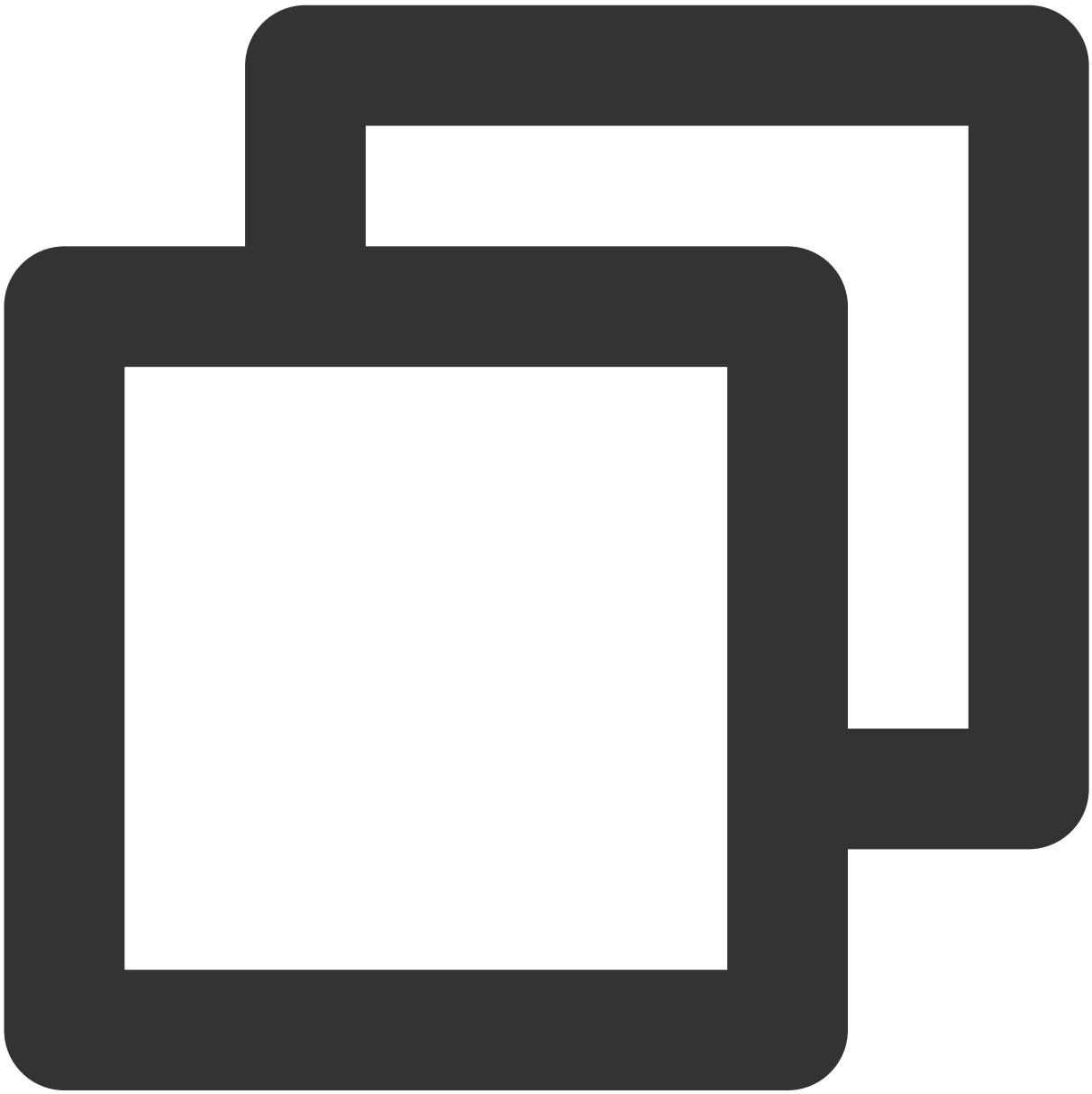


```
package main

import (
    "context"
    "errors"
    "fmt"
    "log"
    "net"
    "net/http"
    "os"
    "os/signal"
```

```
    "time"
    "go.opentelemetry.io/contrib/instrumentation/net/http/otelhttp"
)

func main() {
    if err := run(); err != nil {
        log.Fatalln(err)
    }
}

func run() (err error) {
    ctx, stop := signal.NotifyContext(context.Background(), os.Interrupt)
    defer stop()

    // Initialize SDK
    otelShutdown, err := setupOTelSDK(ctx)
    if err != nil {
        return
    }
    // Graceful shutdown
    defer func() {
        err = errors.Join(err, otelShutdown(context.Background()))
    }()

    // Start HTTP service
    srv := &http.Server{
        Addr:         ":8080",
        BaseContext: func(_ net.Listener) context.Context { return ctx },
        ReadTimeout:  time.Second,
        WriteTimeout: 10 * time.Second,
        Handler:      newHTTPHandler(),
    }
    srvErr := make(chan error, 1)
    go func() {
        srvErr <- srv.ListenAndServe()
    }()

    select {
    case err = <-srvErr:
      return
    case <-ctx.Done():
        stop()
    }

    err = srv.Shutdown(context.Background())
    return
}
```

If implementing an HTTP service through frameworks like Gin, the Event Tracking method will differ. See the community's Framework List for details on other frameworks's Event Tracking methods.

Step 3: Enhanced Event Tracking for HTTP APIs.



```
func newHTTPHandler() http.Handler {
   mux := http.NewServeMux()

   handleFunc := func(pattern string, handlerFunc func(http.ResponseWriter, *http.R
      // HTTP routes Event Tracking
```

```
        handler := otelhttp.WithRouteTag(pattern, http.HandlerFunc(handlerFunc))
        mux.Handle(pattern, handler)
    }

    // Register APIs
    handleFunc("/simple", simpleIOHandler)

    // Enhanced Event Tracking for all APIs
    handler := otelhttp.NewHandler(mux, "/")
    return handler
}

func simpleIOHandler(w http.ResponseWriter, r *http.Request) {
    io.WriteString(w, "ok")
}
```

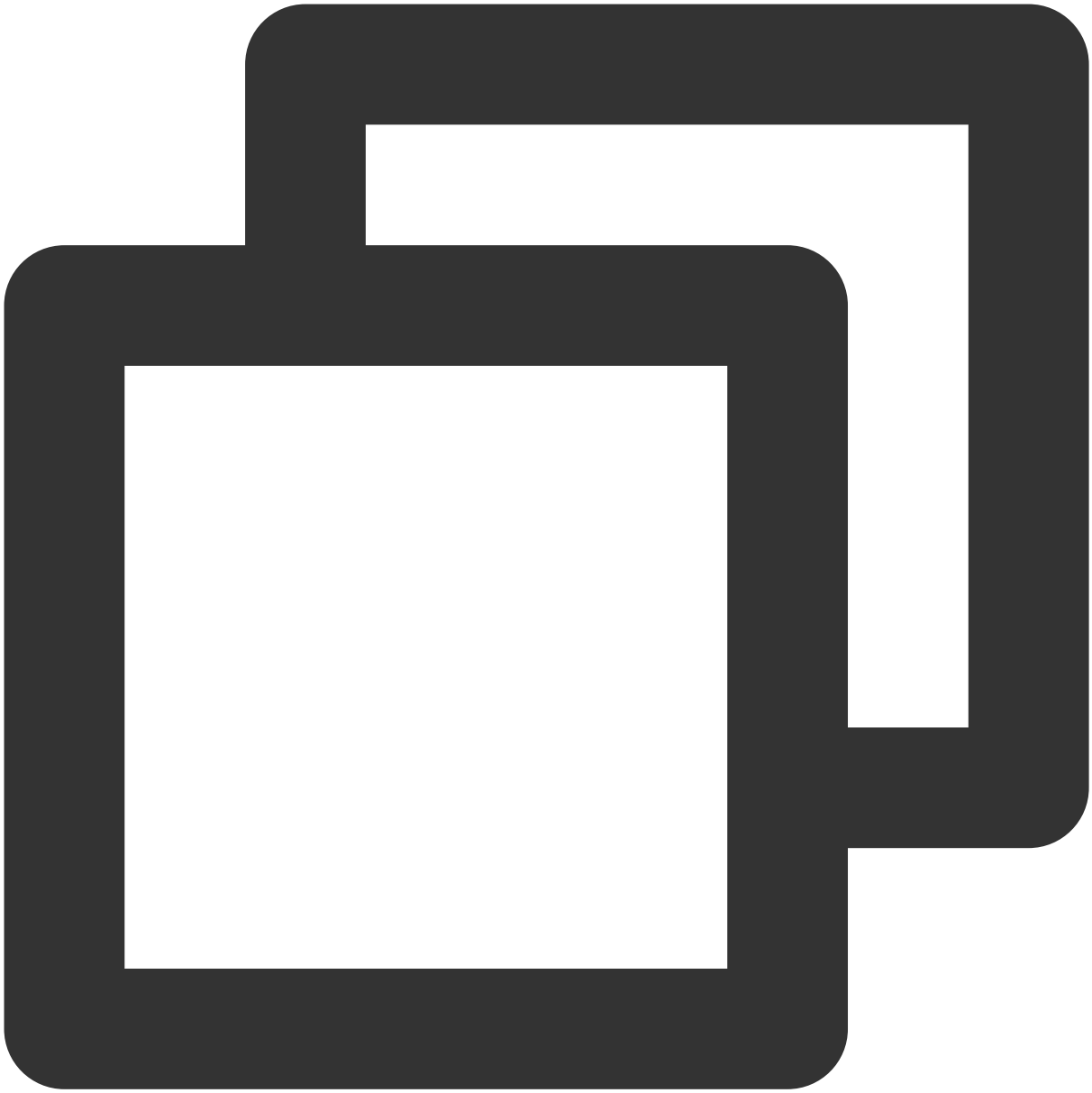
Connection Verification

After you start the Go application, access the corresponding API through port 8080, for example, `https://localhost:8080/simple` , the application reports HTTP request-related link data to APM. In normal traffic cases, the connected applications will be displayed in APM > Application monitoring > Application list and the connected application instances will be displayed in **APM > Application monitoring > Application details > Instance monitoring**. Since there is a certain latency in the processing of observable data, if the application or instance does not appear in the console after connecting, wait for about 30 seconds.

More Event Tracking Sample

Accessing Redis

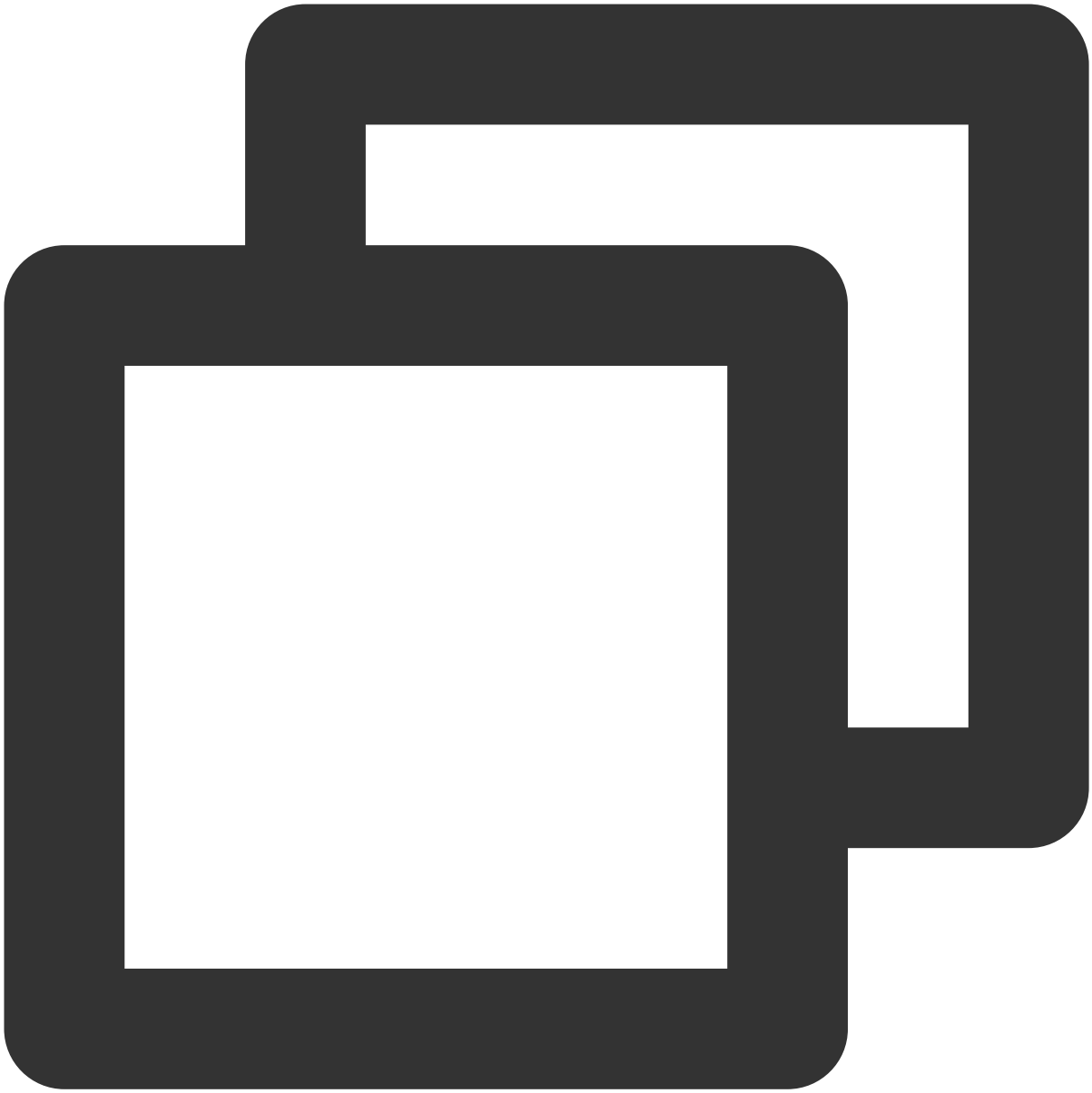
Initialization



```
import (
        "github.com/redis/go-redis/v9"
        "github.com/redis/go-redis/extra/redisotel/v9"
)

var rdb *redis.Client

// InitRedis initializing Redis client.
func InitRedis() *redis.Client {
        rdb := redis.NewClient(&redis.Options{
                Addr:     "127.0.0.1:6379",
```

```
                Password: "",  // no password
        })
        if err := redisotel.InstrumentTracing(rdb); err != nil {
                panic(err)
        }
        if err := redisotel.InstrumentMetrics(rdb); err != nil {
                panic(err)
        }
        return rdb
}
```

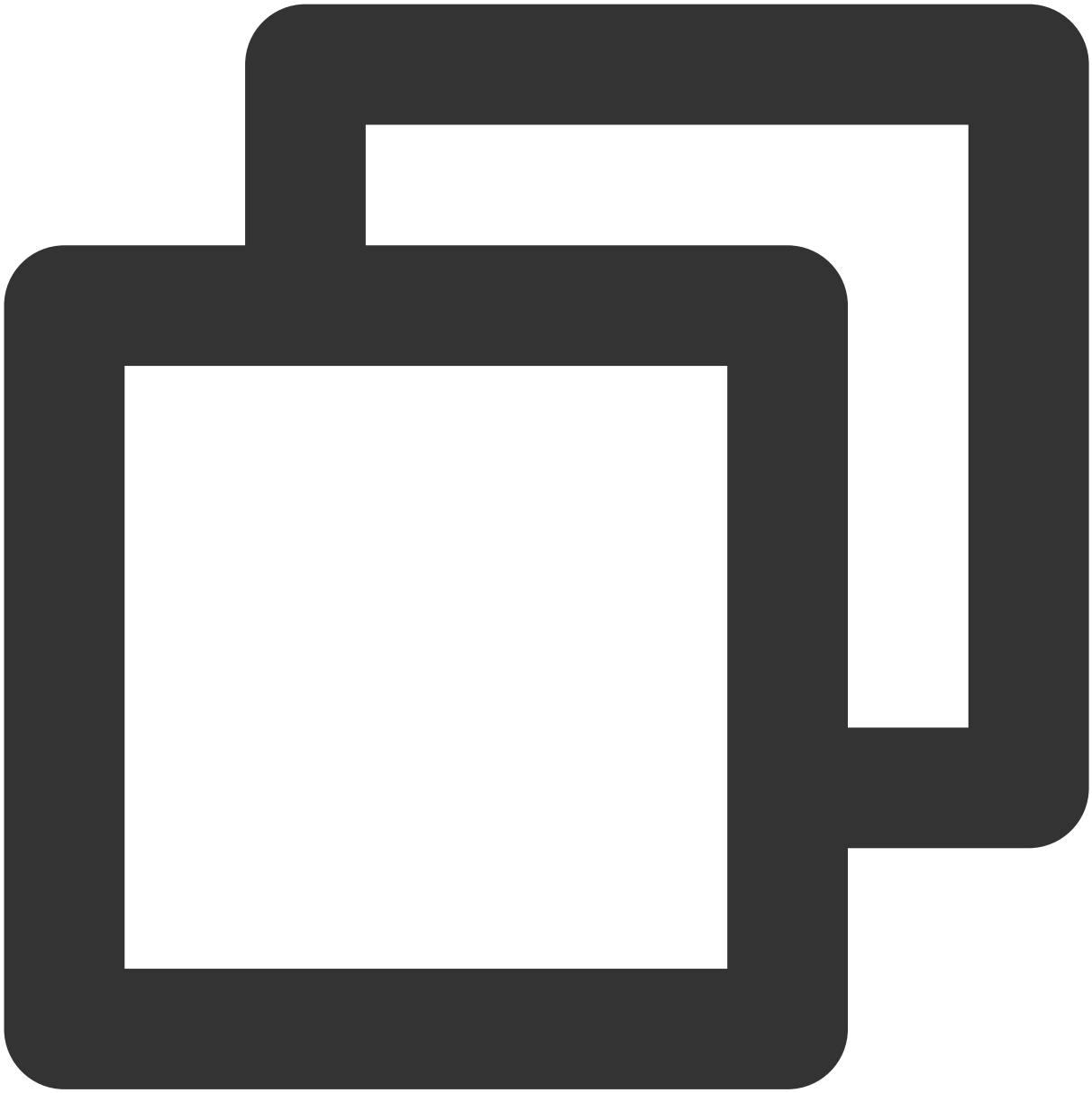
Data Access



```
func redisRequest(w http.ResponseWriter, r *http.Request) {
        ctx := r.Context()
        rdb := InitRedis()
        val, err := rdb.Get(ctx, "foo").Result()
        if err != nil {
                log.Printf("redis err......")
                panic(err)
        }
        fmt.Println("redis res: ", val)
}
```

Accessing MySQL

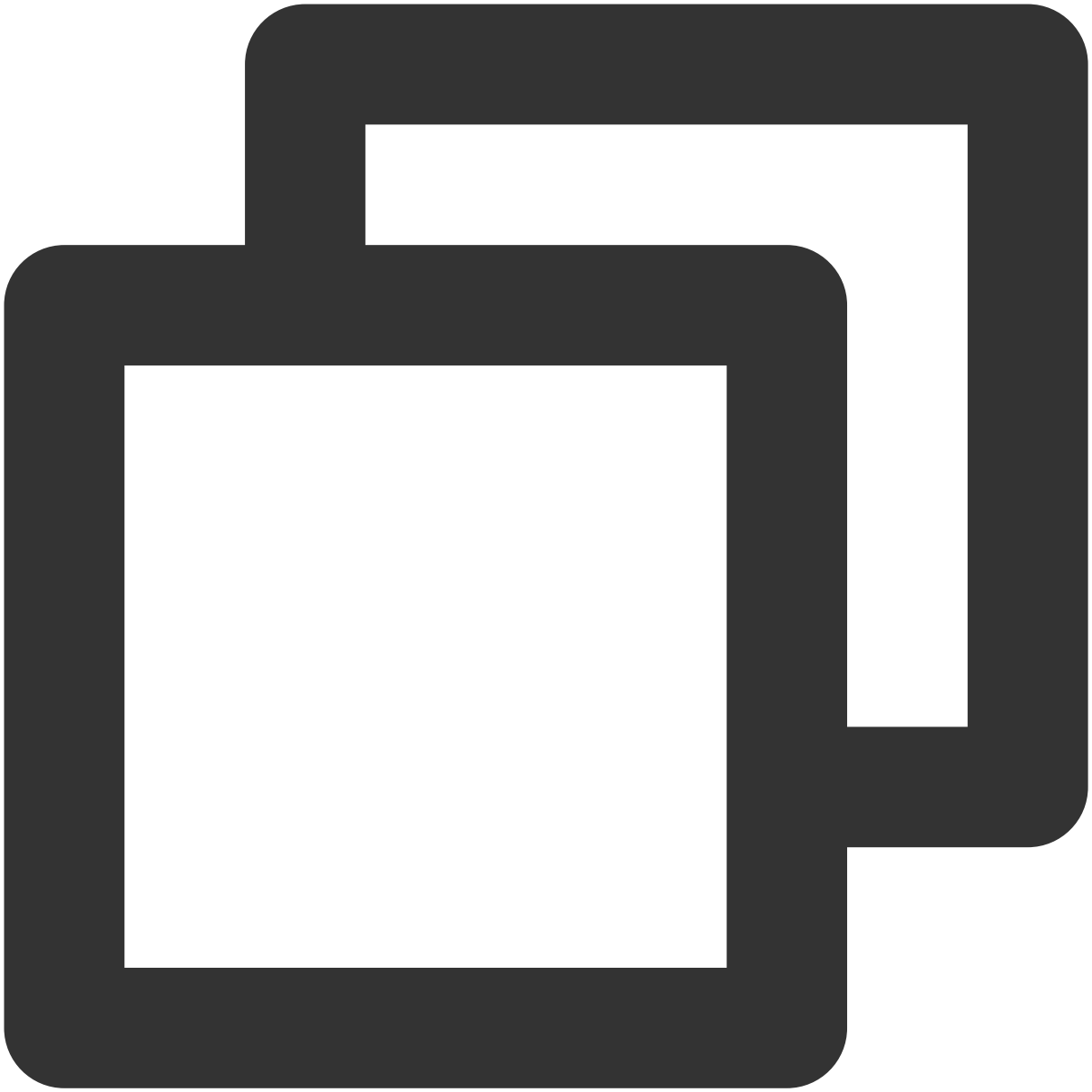
Initialization



```
import (
        "gorm.io/driver/mysql"
        "gorm.io/gorm"
        "gorm.io/gorm/schema"
        "gorm.io/plugin/opentelemetry/tracing"
)

var GormDB *gorm.DB
```

```
type TableDemo struct {
        ID    int    gorm:"column:id"
        Value string gorm:"column:value"
}

func InitGorm() {
        var err error

        dsn := "root:4T$er3deffYuD#9Q@tcp(127.0.0.1:3306)/db_demo?charset=utf8mb4&p
        GormDB, err = gorm.Open(mysql.Open(dsn), &gorm.Config{
                NamingStrategy: schema.NamingStrategy{
                        SingularTable: true, // Use singular table names.
                },
        })
        if err != nil {
                panic(err)
        }
    // Add tracing reporting logic.
    //Fill in DBName based on actual conditions. In the APM topology diagram, ident
        if err = GormDB.Use(tracing.NewPlugin(tracing.WithoutMetrics(),tracing.With
                panic(err)
        }
}
```

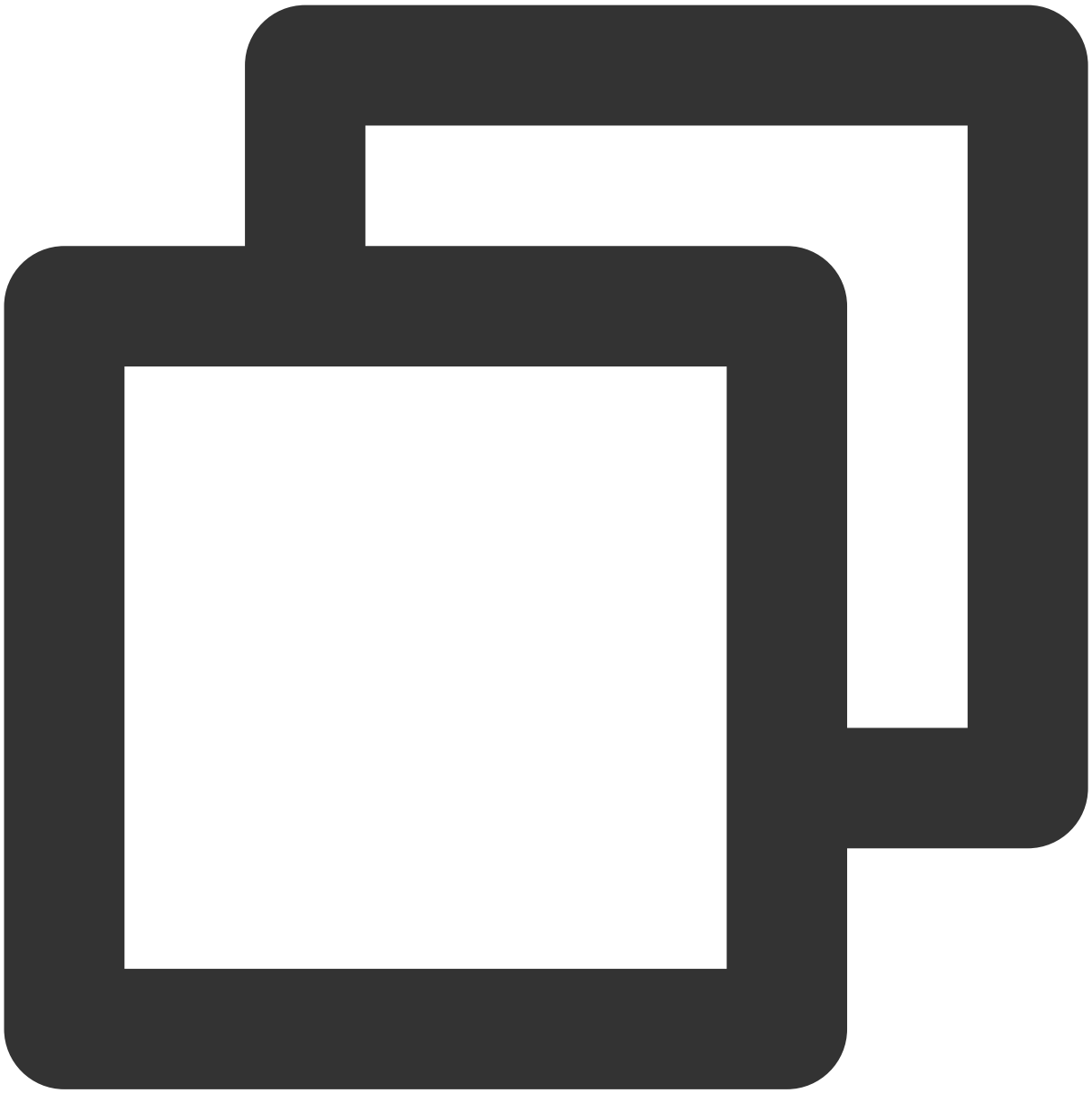
Data Access



```
func gormRequest(ctx context.Context) {
    var val string
    if err := gormclient.GormDB.WithContext(ctx).Model(&gormclient.TableDemo{}).Whe
            panic(err)
    }
    fmt.Println("MySQL query result: ", val)
}
```

Internal method Event Tracking and setting Span attributes.

The following code demonstrates internal method Event Tracking by inserting an Internal type Span into the current link context.



```
func internalSpanFunc(w http.ResponseWriter, r *http.Request) {
        internalInvoke(r)
        io.WriteString(w, "ok")
}

func internalInvoke(r *http.Request) {
    // Create an Internal Span.
        _, span := tracer.Start(r.Context(), "internalInvoke")
```

```
        defer span.End()

    // Business logic is omitted.

    // Set Span Attributes.
        span.SetAttributes(attribute.KeyValue{
                Key:   "label-key-1",
                Value: attribute.StringValue("label-value-1"),
        })
}
```

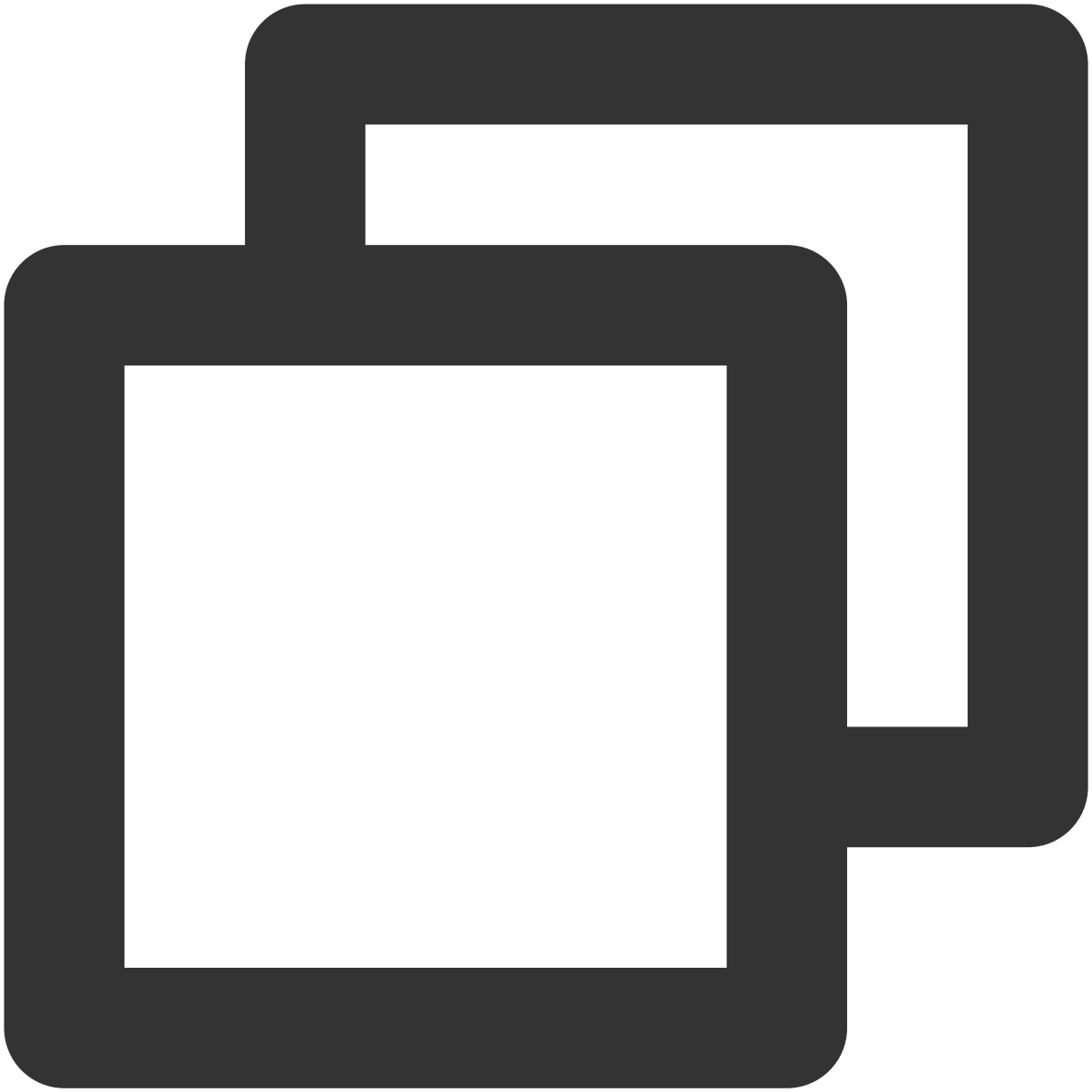
Connecting Go Applications Using SkyWalking-Go

Last updated：2024-06-19 16:31:30

SkyWalking Go is a performance monitoring scheme for Go applications provided by the SkyWalking community. It allows Go applications to connect to APM without altering business code. For more information on SkyWalking Go, see the Project Documentation. SkyWalking Go supports automatic Event Tracking for commonly used Go dependency libraries and frameworks, including Gin, GORM, gRPC, etc. For other libraries and frameworks supporting automatic Event Tracking, see the Complete List provided by the SkyWalking community.

Demo Applications

With the following demo codes, you can start the simplest HTTP service:



```
package main

import (
        "net/http"
)
func main() {
        http.HandleFunc("/hello", func(writer http.ResponseWriter, request *http.Re
                writer.Write([]byte("Hello World from skywalking-go-agent"))
        })
        err := http.ListenAndServe(":8080", nil)
        if err != nil {
```

```
                panic(err)
        }
}
```

Preliminary steps: Get the connect point and Token.

1. Log in to the TCOP console.
2. In the left menu column, select **Application Performance Management > Application monitoring**, and click **Application list > Access application**.
3. In the **Data access** drawer that pops up on the right, click the Go language.
4. On the **Access Go application** page, select the **Region** and **Business System** you want to connect.
5. Choose the **Access protocol type** as **SkyWalking**.
6. **Reporting method** Choose your desired reporting method, and obtain your **Access Point** and **Token**.

Note:

Private network reporting: Using this reporting method, your service needs to run in the Tencent Cloud VPC. Through VPC connecting directly, you can avoid the security risks of public network communication and save on reporting traffic overhead.

Public network reporting: If your service is deployed locally or in non-Tencent Cloud VPC, you can report data in this method. However, it involves security risks in public network communication and incurs reporting traffic fees.

Connecting Go Applications

Step 1. Download the Agent.

Go to the SkyWalking Download Page, in the Go Agent section, click Distribution to download the tar format of the Agent packet, with the file name suffix as `tgz` .

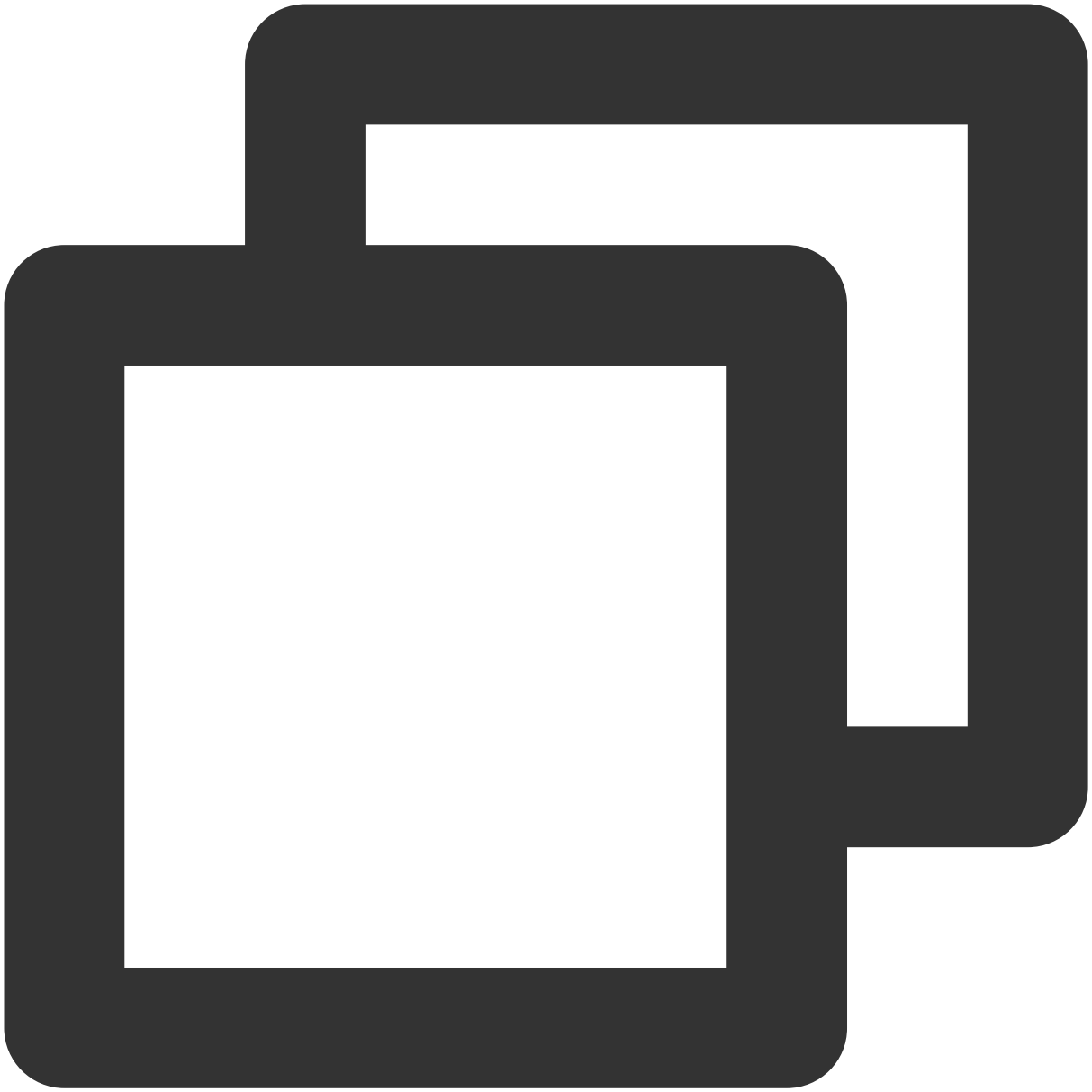
After the packet is extracted, you obtain the binary files under the `bin` directory. Choose the binary file that matches your operating system as the Agent file. For example, in the Linux system, the Agent file is `skywalking-go-agent-0.4.0-linux-amd64` .

Step 2: Install the Agent.

SkyWalking Go provides 2 methods to install the Agent, you can choose either method:

Agent Injection Method

If you do not need to customize Event Tracking in the code, you can choose the Agent injection method. Execute the command as follows:

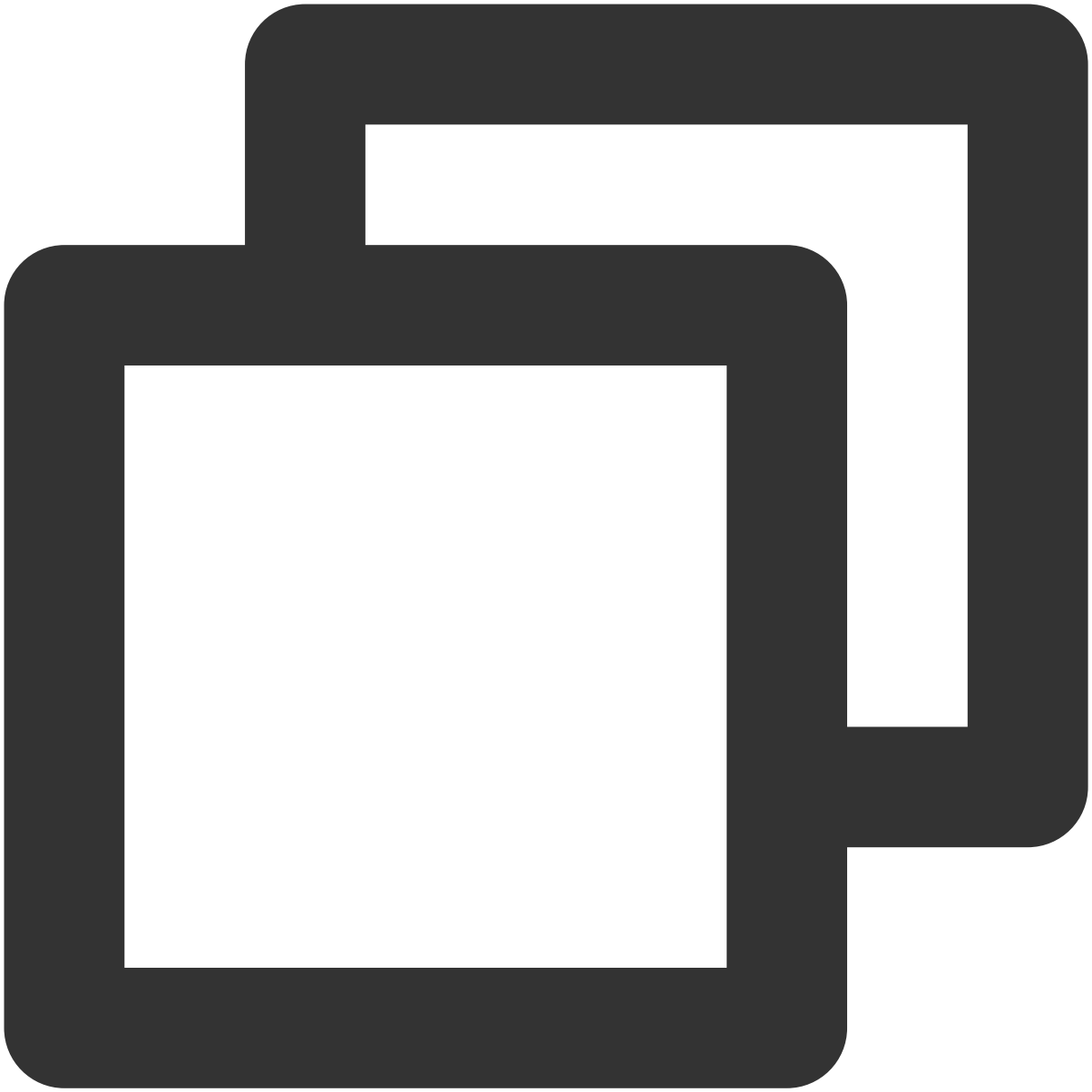


```
/path/to/agent -inject /path/to/your/project [-all]
```

Here, `/path/to/agent` is the Agent file obtained in step 1, `/path/to/your/project` is the Go project root directory.

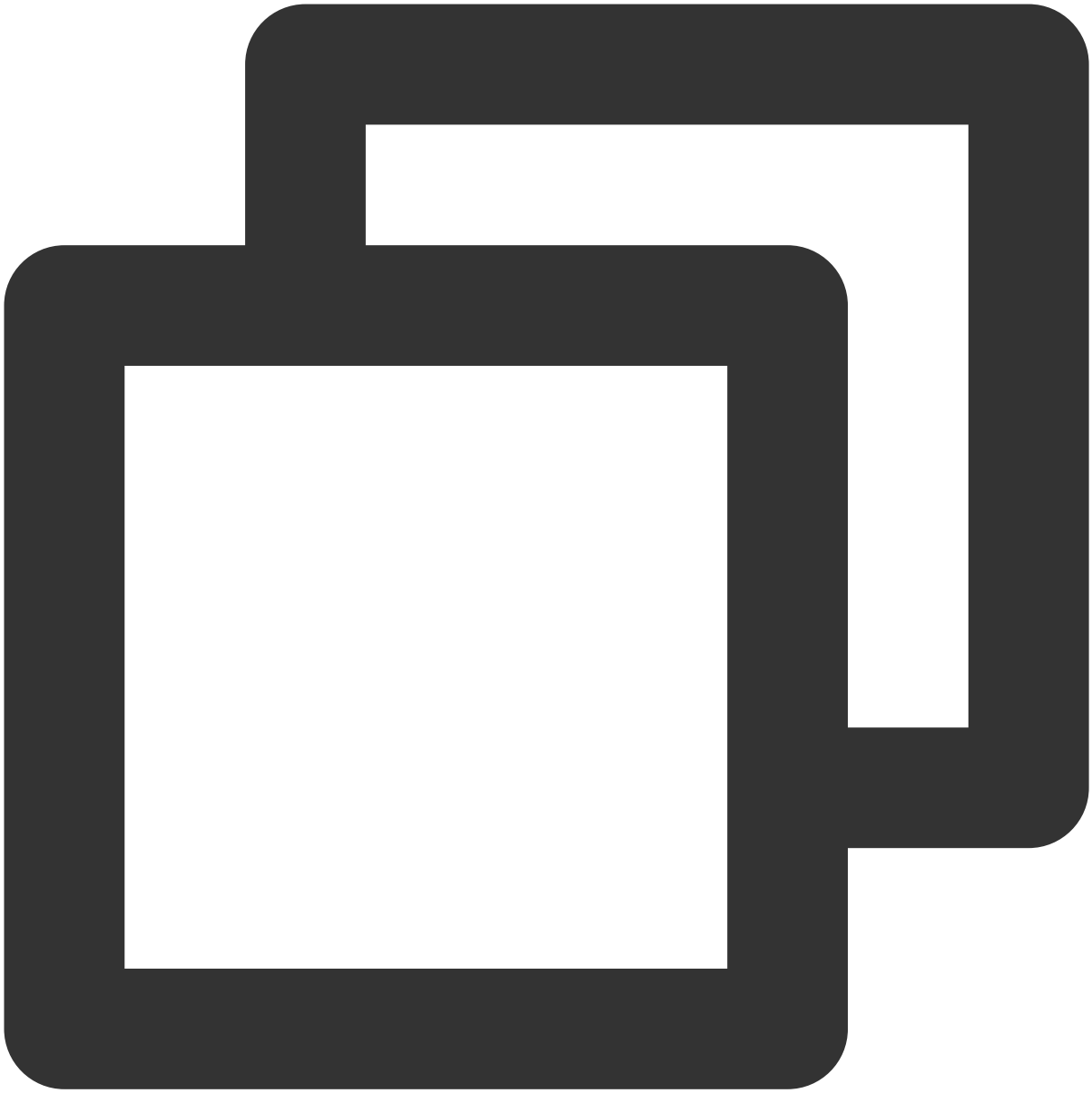
Code Dependency Method

Run the following command to obtain the required dependencies:



```
go get github.com/apache/skywalking-go
```

Include the dependency in main:

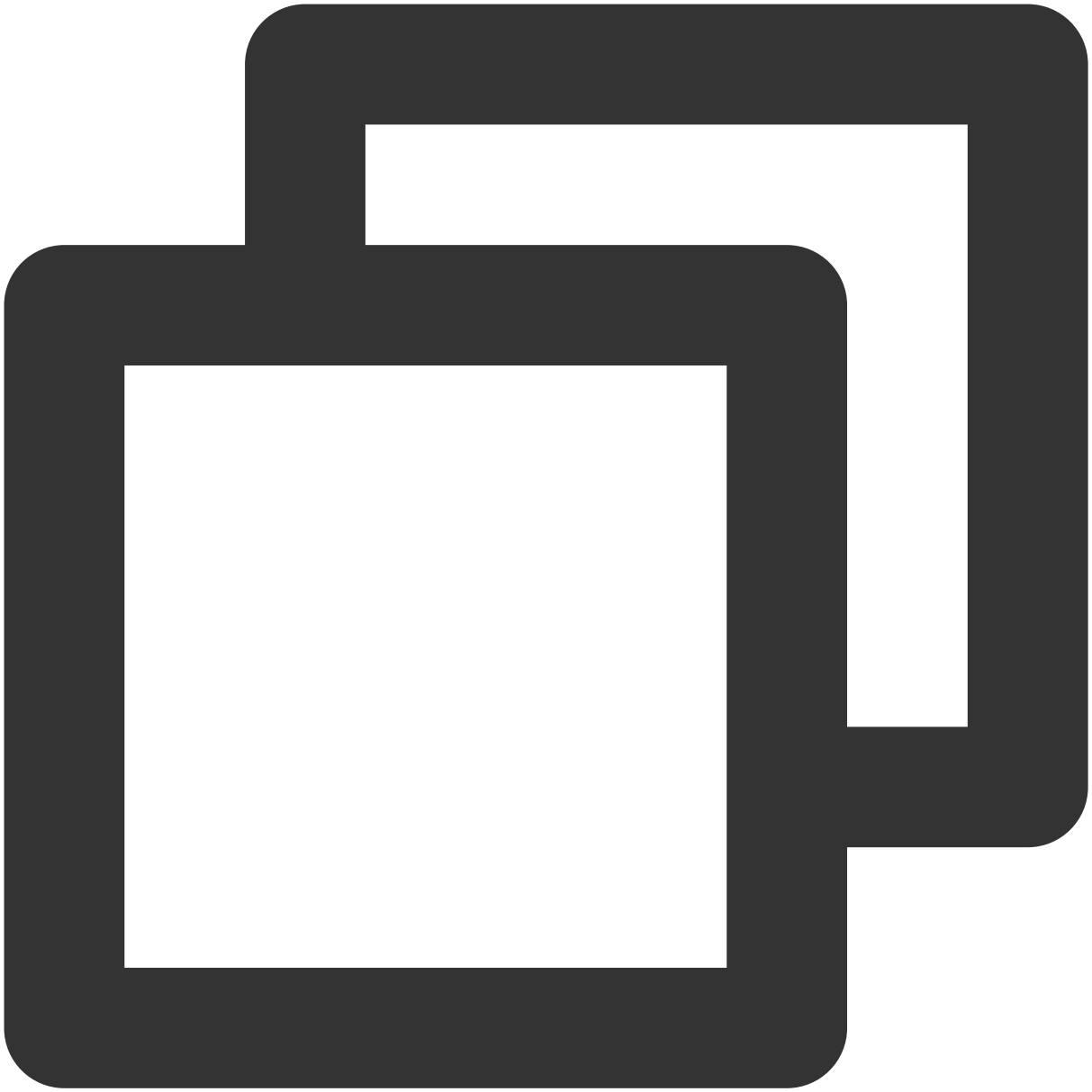


```
import _ "github.com/apache/skywalking-go"
```

Step 3: Modify the configuration for connecting APM.

Obtain the configuration file template from the community's default configuration file, and save it as a text file, which can be named `config.yaml` .

Modify the configuration file, at least the following 3 items need to be configured:



```
agent:
  service_name: "<serviceName>" # Replace <serviceName> with the application name.
reporter:
  grpc:
    backend_service: "<endpoint>" # Replace <endpoint> with the reporting address.
    authentication: "<token>" # Replace <token> with the business system Token.
```

The corresponding field descriptions are as follows:

`<serviceName>` : Application name. Multiple application processes connecting with the same serviceName are displayed as multiple instances under the same application in APM. The application name can be up to 63 characters

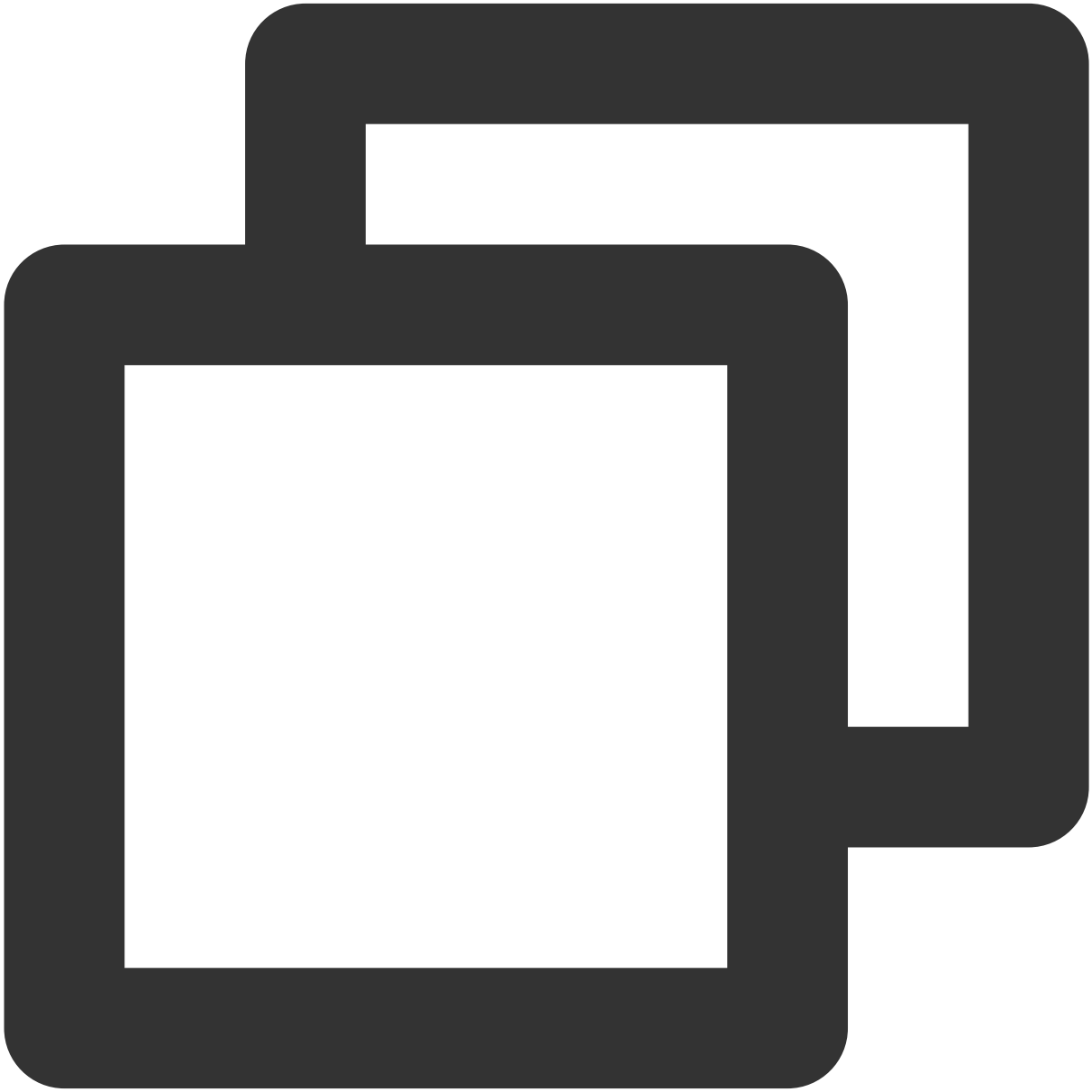
and can only contain lowercase letters, digits, and the separator (-), and it must start with a lowercase letter and end with a digit or lowercase letter.

`<token>` : The business system Token obtained in the preliminary steps.

`<endpoint>` : The connect point obtained in the preliminary steps.

Step 4: Compile projects based on SkyWalking-Go.

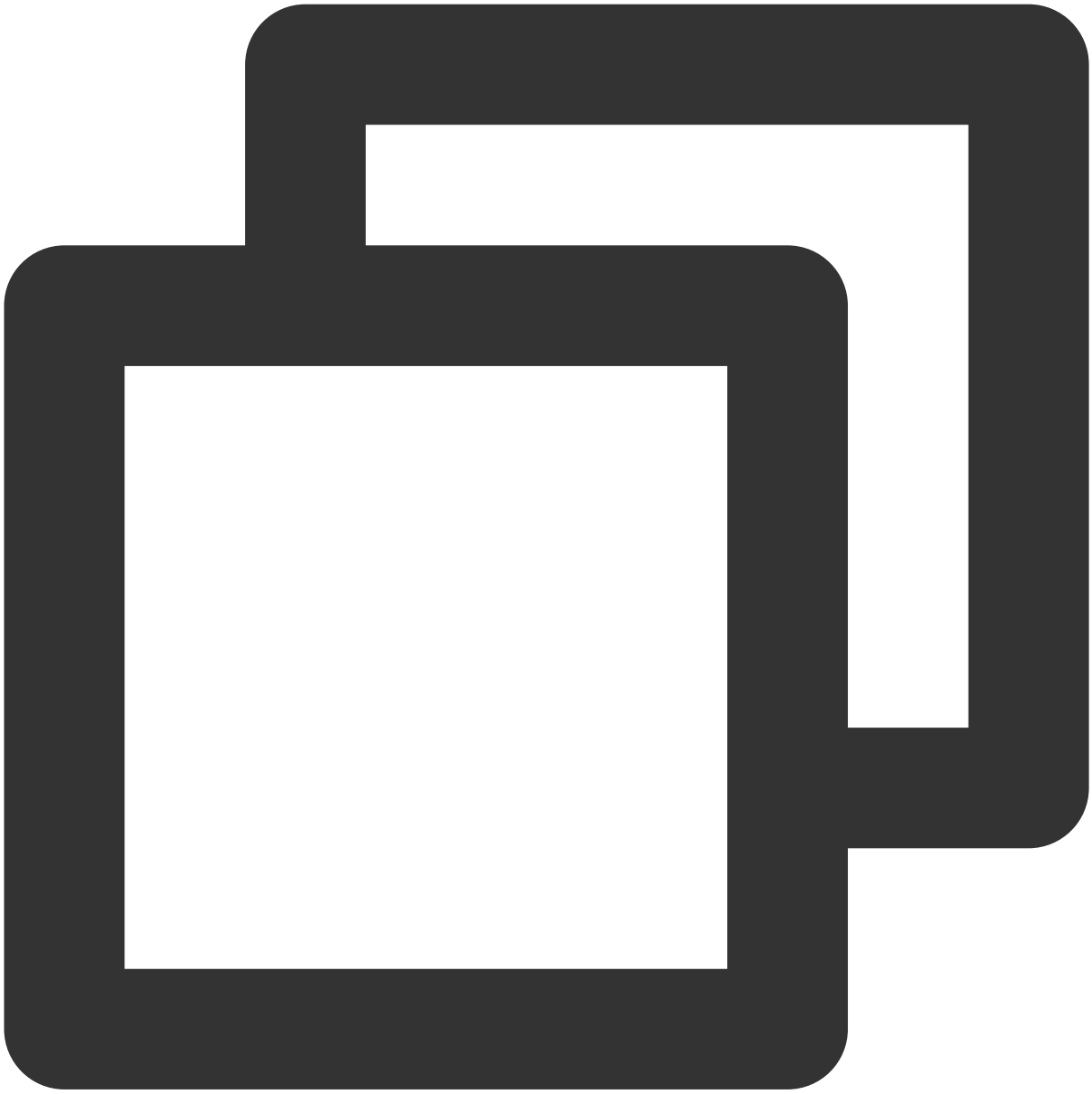
When you compile the Go project, add the following parameters:



```
-toolexec="/path/to/agent" -config /path/to/config.yaml -a
```

Where, `/path/to/agent` is the Agent file obtained in step 1, `/path/to/config.yaml` is the configuration file obtained in step 3.

Assuming the compiled output is named test, the complete command is:



```
go build -toolexec='/path/to/agent -config /path/to/config.yaml' -a -o test .
```

Connection Verification

After you start the Go application, access the corresponding API through port 8080, for example, `https://localhost:8080/hello` , the application reports the HTTP request-related link data to APM. In normal traffic cases, the connected application will displayed in APM > Application monitoring > Application list and the connected application instances will be displayed in **APM > Application monitoring > App details > Instance monitoring**. Since there is a certain latency in the processing of observable data, if the application or instance does not appear in the console after connecting, wait for about 30 seconds.

Custom Link Event Tracking

When automatic Event Tracking does not meet your needs, or you need to add business layer instrumentation, see the community's Tracing API Documentation and add custom link instrumentation in the code.

Reporting over Jaeger Protocol
Reporting with Native Jaeger SDK

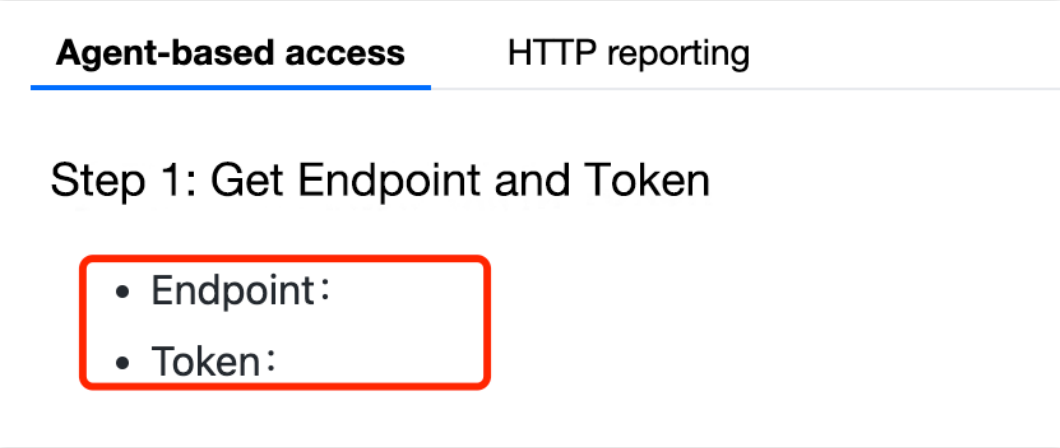
Last updated：2023-12-25 15:59:08

This document describes how to report the data of a Go application with the native Jaeger SDK.

Directions

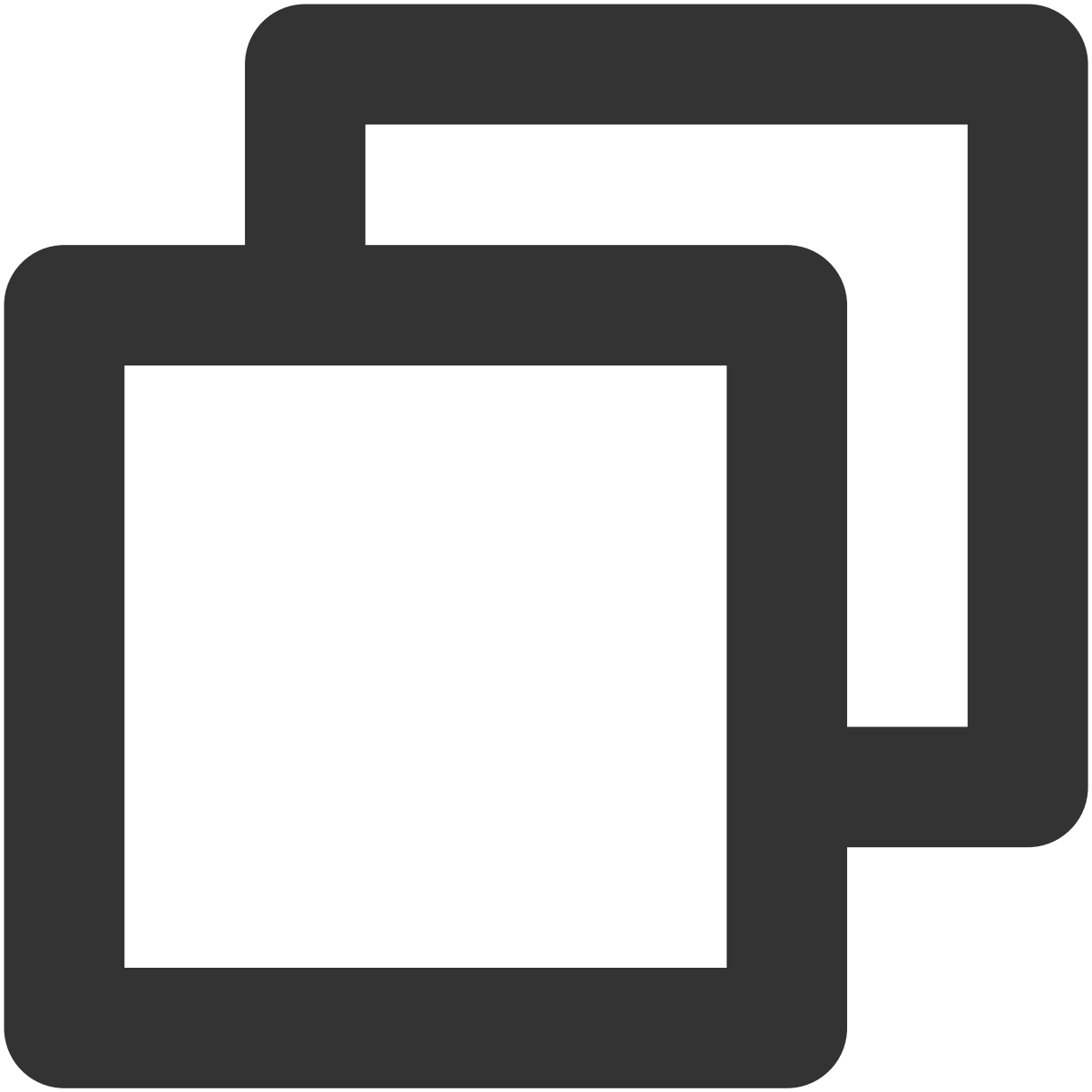
Step 1. Get the endpoint and token

Log in to the APM console, enter the **Application monitoring** > **Application list** page, click **Access application**, and select the Go language and the native Jaeger SDK data collection method.
Then, get the endpoint and token in the step of access method selection.



Step 2. Install the Jaeger agent

1. Download the Jaeger agent.
2. Run the following command to start the agent:



```
nohup ./jaeger-agent --reporter.grpc.host-port={{endpoint}} --agent.tags=token={{to
```

Note:

For Jaeger agent v1.15.0 and earlier, replace `--agent.tags` in the startup command with `--jaeger.tags`.

Step 3. Report data

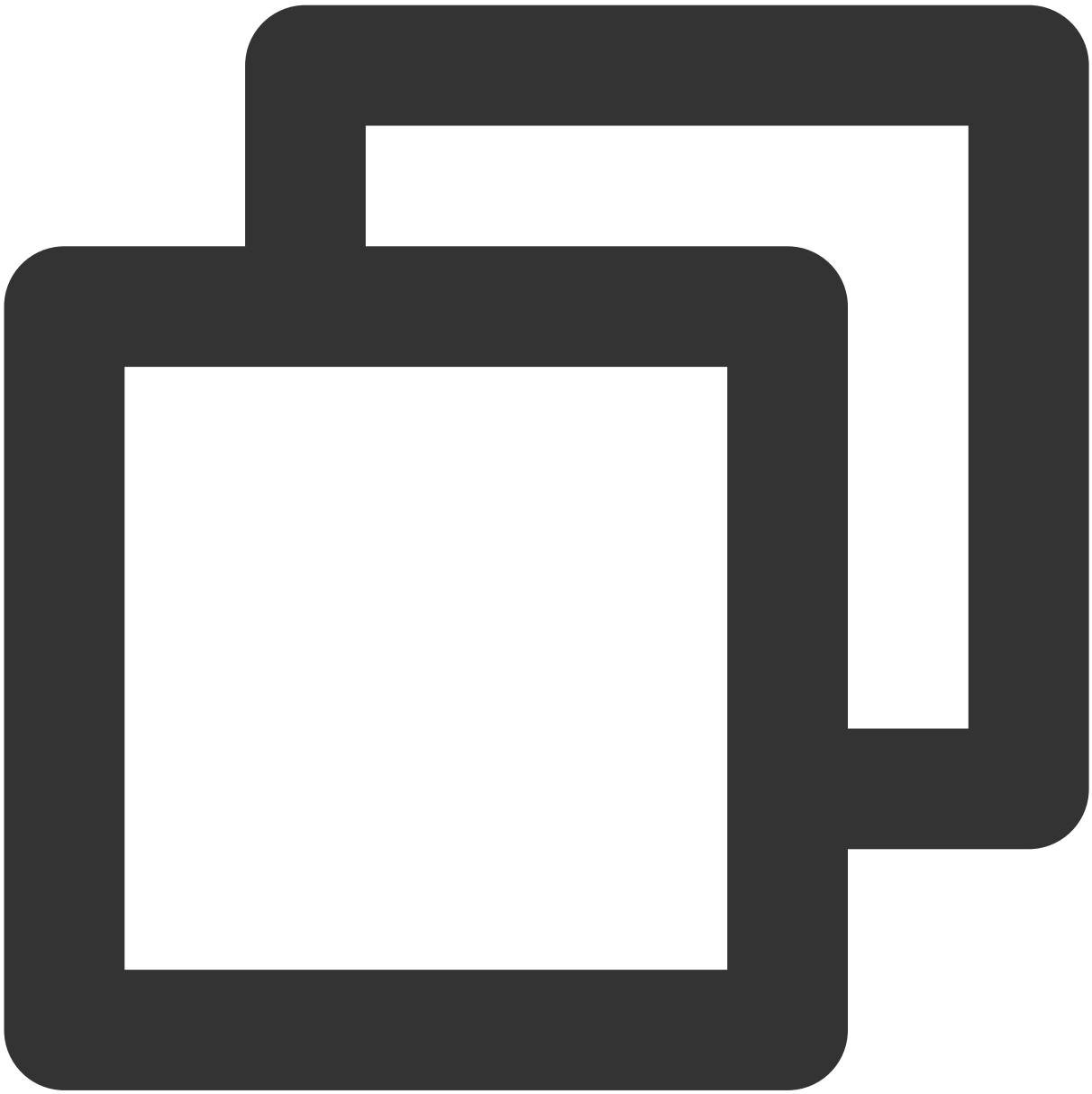
Report data through the native Jaeger SDK:

1. Due to the need to simulate HTTP requests on the client, import the `opentracing-contrib/go-stdlib/nethttp` dependency.

Dependency path: `github.com/opentracing-contrib/go-stdlib/nethttp`

Version requirement: ≥ `dv1.0.0`

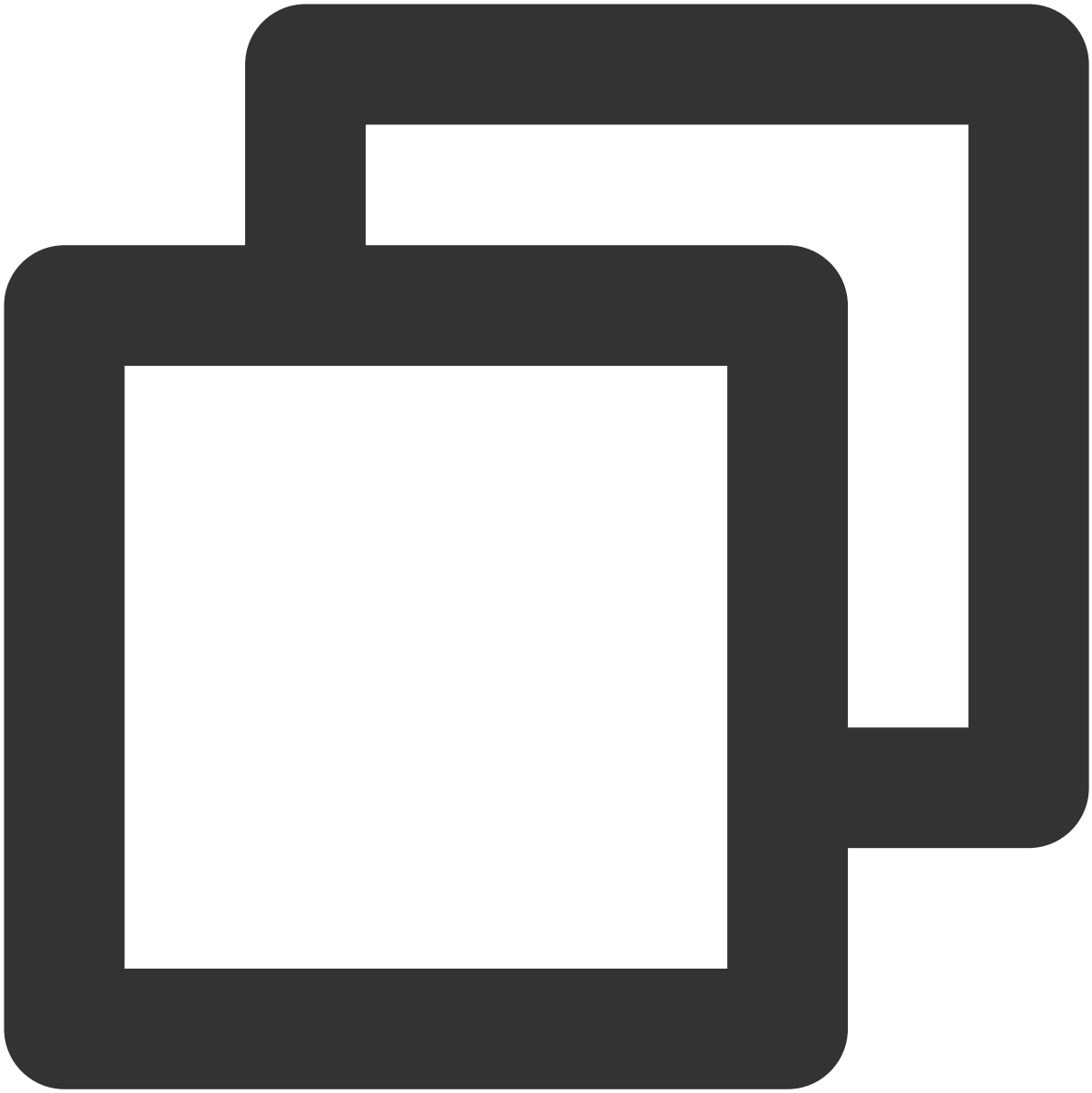
2. Configure Jaeger and create a trace object. Below is a sample:



```
cfg := &jaegerConfig.Configuration{
    ServiceName: ginClientName, // Call trace of the target service. Enter the serv
    Sampler: &jaegerConfig.SamplerConfig{ // Sampling policy configuration. See sec
    Type:  "const",
    Param: 1,
    },
    Reporter: &jaegerConfig.ReporterConfig{ // Configure how the client reports tra
```

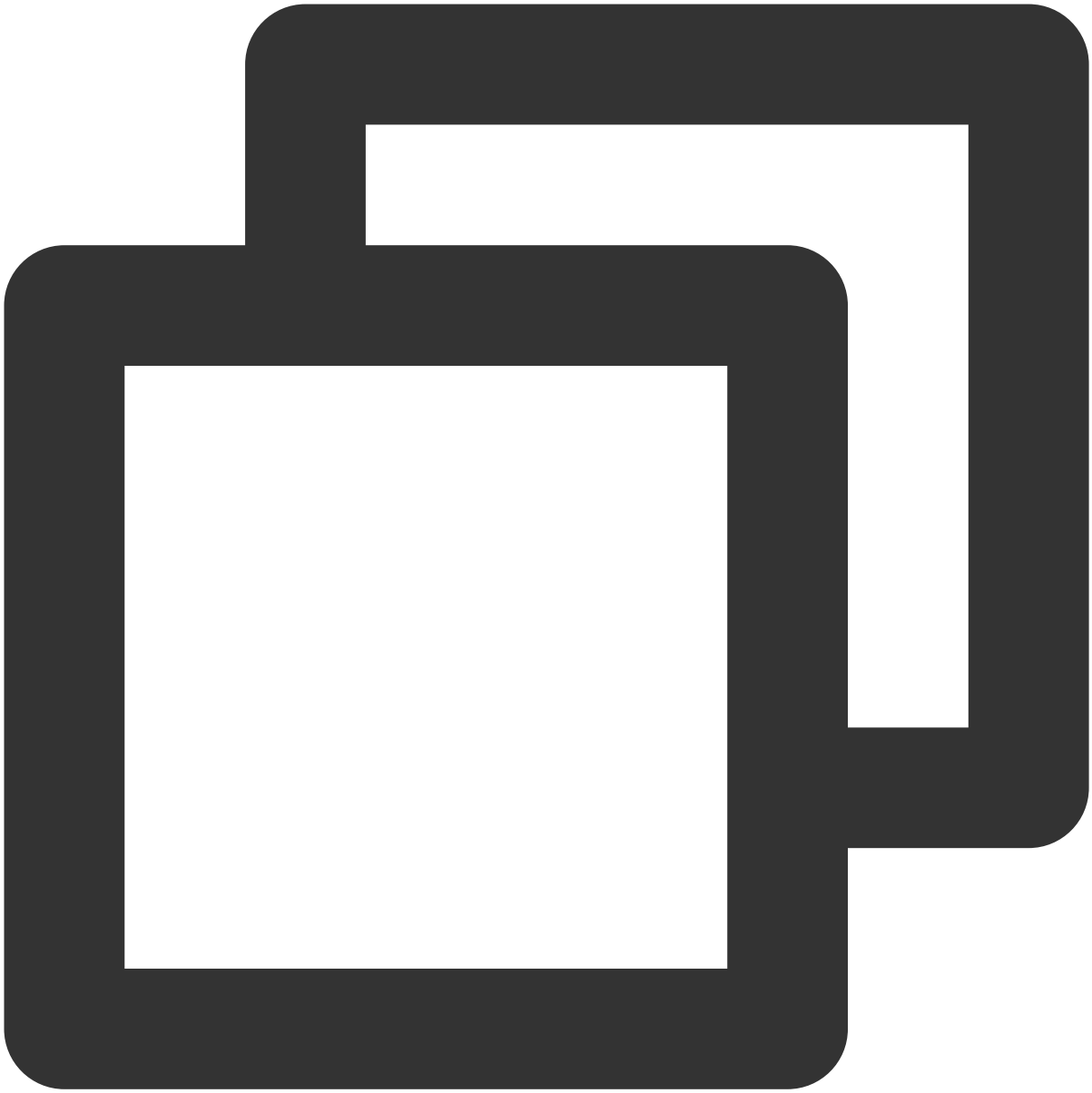
```
    LogSpans:          true,
    LocalAgentHostPort: endPoint,
    },
    // Token configuration
    Tags:        []opentracing.Tag{ // Set the tag, where information such as token
    opentracing.Tag{Key: "token", Value: token}, // Set the token
    },
}

tracer, closer, err := cfg.NewTracer(jaegerConfig.Logger(jaeger.StdLogger)) // Get
```

3. Construct a span and put it in the context. Below is a sample:



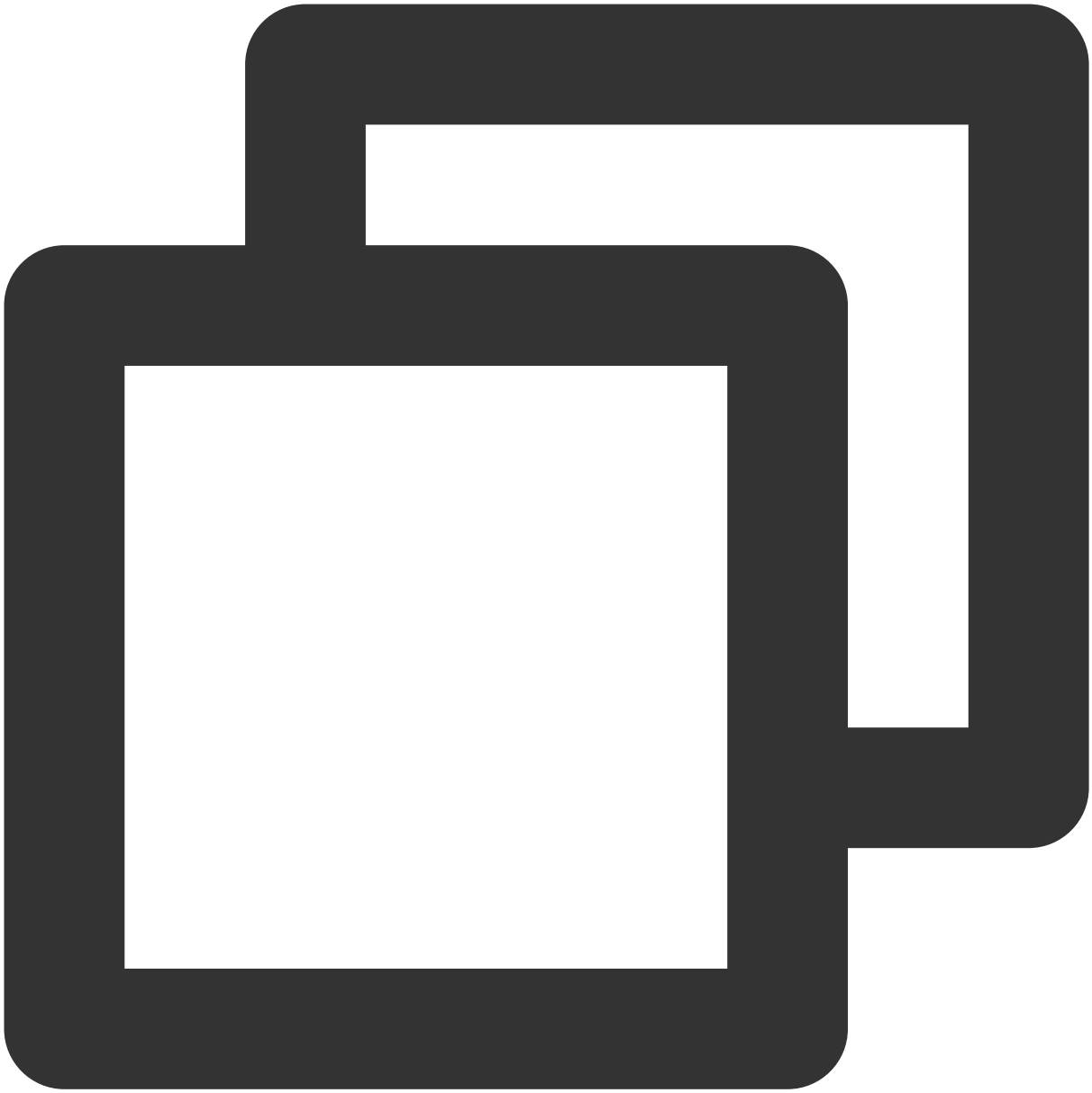
```
span := tracer.StartSpan("CallDemoServer") // Construct a span
ctx := opentracing.ContextWithSpan(context.Background(), span) // Put the span refe
```

4. Construct a request with the tracer. Below is a sample:



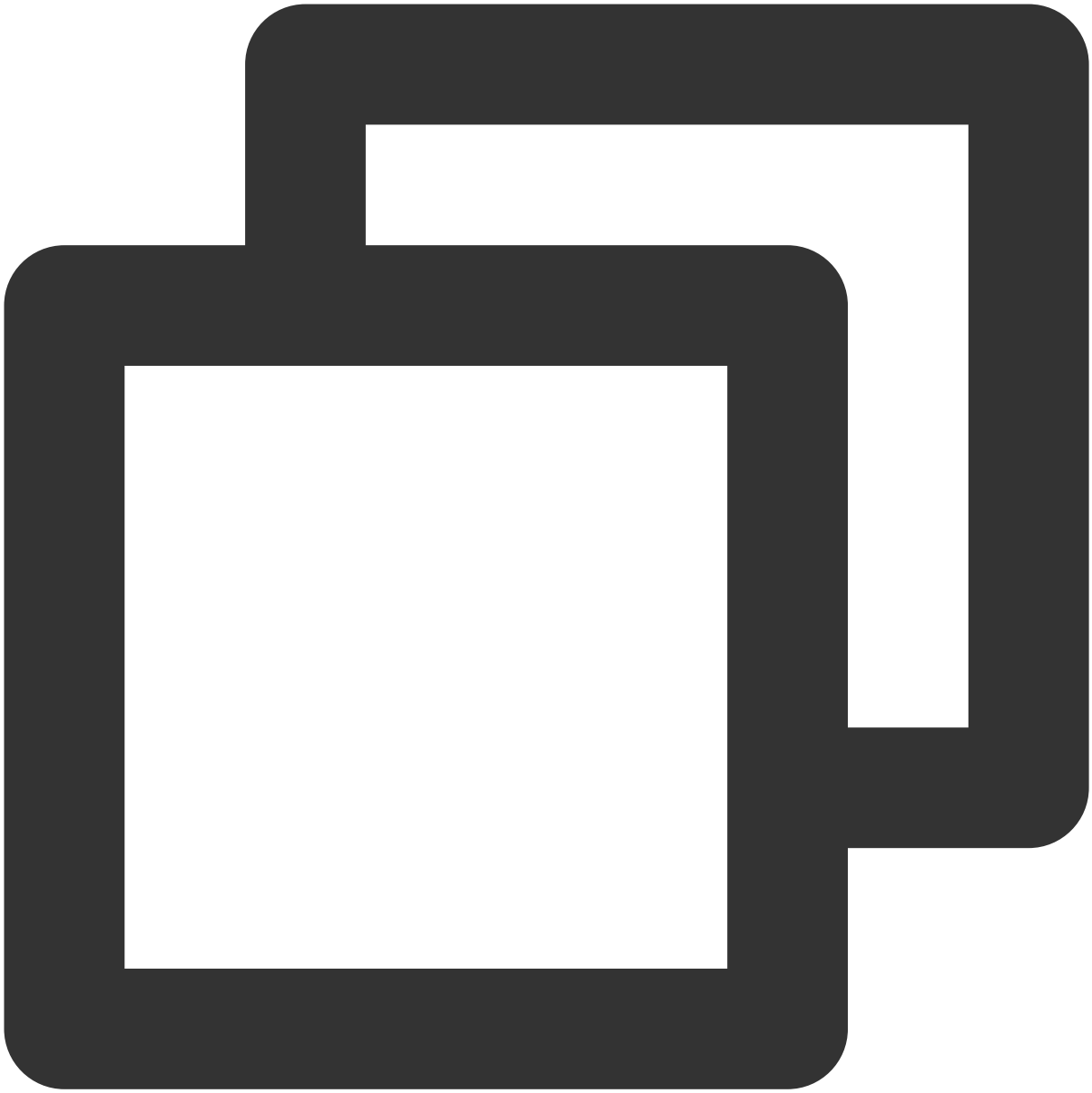
```
// Construct an HTTP request
req, err := http.NewRequest(
        http.MethodGet,
        fmt.Sprintf("http://localhost%s/ping", ginPort),
        nil,
    )
req = req.WithContext(ctx)
// Construct a request with the tracer
req, ht := nethttp.TraceRequest(tracer, req)
```

5. Send an HTTP request and get the returned result.



```
httpClient := &http.Client{Transport: &nethttp.Transport{}} // Initialize the HTTP
res, err := httpClient.Do(req)
// ... Error judgment is omitted.
body, err := ioutil.ReadAll(res.Body)
// ... Error judgment is omitted.
log.Printf(" %s recevice: %s\\n", clientServerName, string(body))
```

The complete code is as follows:



```
// Copyright © 2019-2020 Tencent Co., Ltd.

// This file is part of tencent project.
// Do not copy, cite, or distribute without the express
// permission from Cloud Monitor group.

package gindemo

import (
    "context"
    "fmt"
```

```
    "github.com/opentracing-contrib/go-stdlib/nethttp"
    "github.com/opentracing/opentracing-go"
    "github.com/opentracing/opentracing-go/ext"
    opentracingLog "github.com/opentracing/opentracing-go/log"
    "github.com/uber/jaeger-client-go"
    jaegerConfig "github.com/uber/jaeger-client-go/config"
    "io/ioutil"
    "log"
    "net/http"
)

const (
    // Service name, which is the unique identifier of the service and the basis fo
    ginClientName = "demo-gin-client"
    ginPort       = ":8080"
    endPoint      = "xxxxx:6831" // Local agent address
    token         = "abc"
)

// The Gin client under StartClient is also a standard HTTP client.
func StartClient() {
    cfg := &jaegerConfig.Configuration{
        ServiceName: ginClientName, // Call trace of the target service. Enter the
        Sampler: &jaegerConfig.SamplerConfig{ // Sampling policy configuration. See
            Type:  "const",
            Param: 1,
        },
        Reporter: &jaegerConfig.ReporterConfig{ // Configure how the client reports
            LogSpans:           true,
            LocalAgentHostPort: endPoint,
        },
        // Token configuration
        Tags: []opentracing.Tag{ // Set the tag, where information such as token ca
            opentracing.Tag{Key: "token", Value: token}, // Set the token
        },
    }

    tracer, closer, err := cfg.NewTracer(jaegerConfig.Logger(jaeger.StdLogger)) //
    defer closer.Close()
    if err != nil {
        panic(fmt.Sprintf("ERROR: fail init Jaeger: %v\\n", err))
    }
    // Construct a span and put it in the context
    span := tracer.StartSpan("CallDemoServer")
    ctx := opentracing.ContextWithSpan(context.Background(), span)
    defer span.Finish()
```

```
    // Construct an HTTP request
    req, err := http.NewRequest(
        http.MethodGet,
        fmt.Sprintf("http://localhost%s/ping", ginPort),
        nil,
    )
    if err != nil {
        HandlerError(span, err)
        return
    }
    // Construct a request with a tracer
    req = req.WithContext(ctx)
    req, ht := nethttp.TraceRequest(tracer, req)
    defer ht.Finish()

    // Initialize the HTTP client
    httpClient := &http.Client{Transport: &nethttp.Transport{}}
    // Send the request
    res, err := httpClient.Do(req)
    if err != nil {
        HandlerError(span, err)
        return
    }
    defer res.Body.Close()
    body, err := ioutil.ReadAll(res.Body)
    if err != nil {
        HandlerError(span, err)
        return
    }
    log.Printf(" %s recevice: %s\\n", ginClientName, string(body))
}

// HandlerError handle error to span.
func HandlerError(span opentracing.Span, err error) {
    span.SetTag(string(ext.Error), true)
    span.LogKV(opentracingLog.Error(err))
}
```

Reporting with Gin Jaeger Middleware

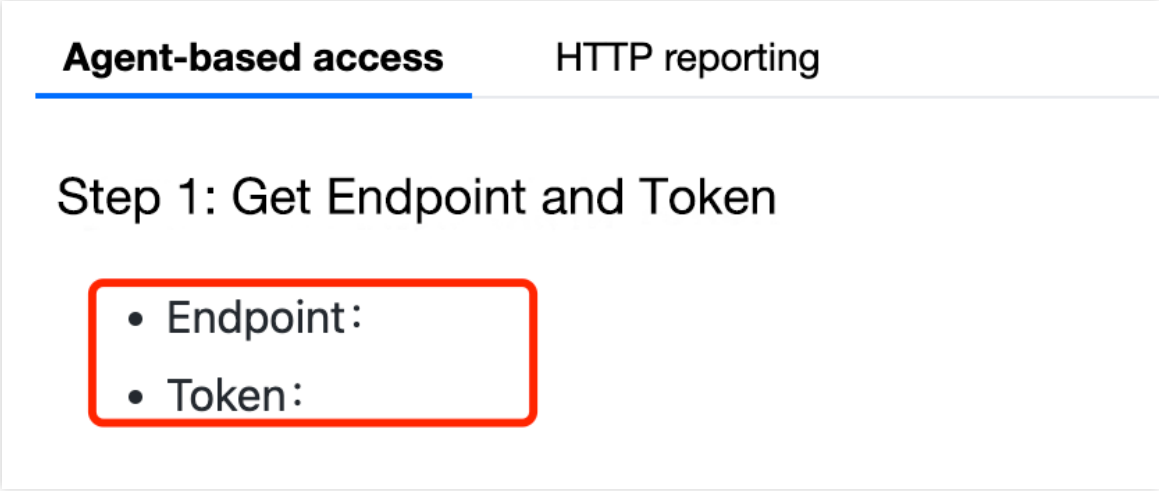
Last updated：2023-12-25 16:00:03

This document describes how to report the data of a Go application with the Gin Jaeger middleware.

Directions

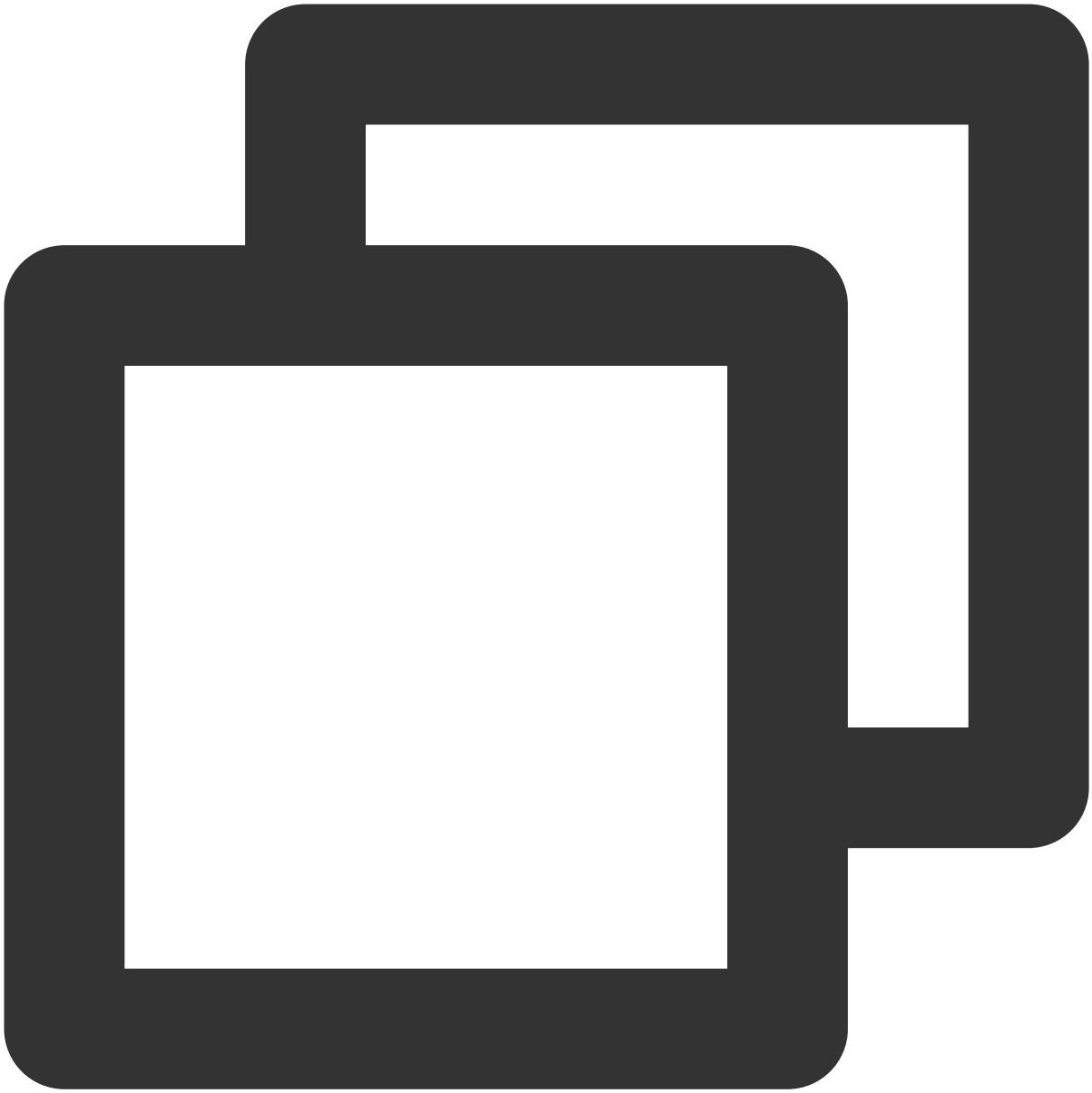
Step 1. Get the endpoint and token

Log in to the APM console, enter the **Application monitoring** > **Application list** page, click **Access application**, and select the Go language and the Gin Jaeger data collection method.
Then, get the endpoint and token in the step of access method selection.



Step 2. Install the Jaeger agent

1. Download the Jaeger agent.
2. Run the following command to start the agent:



```
nohup ./jaeger-agent --reporter.grpc.host-port={{collectorRPCHostPort}} --agent.tag
```

Step 3. Select the reporting type to report application data

Select the reporting type to report the data of a Go application through the Gin Jaeger middleware:

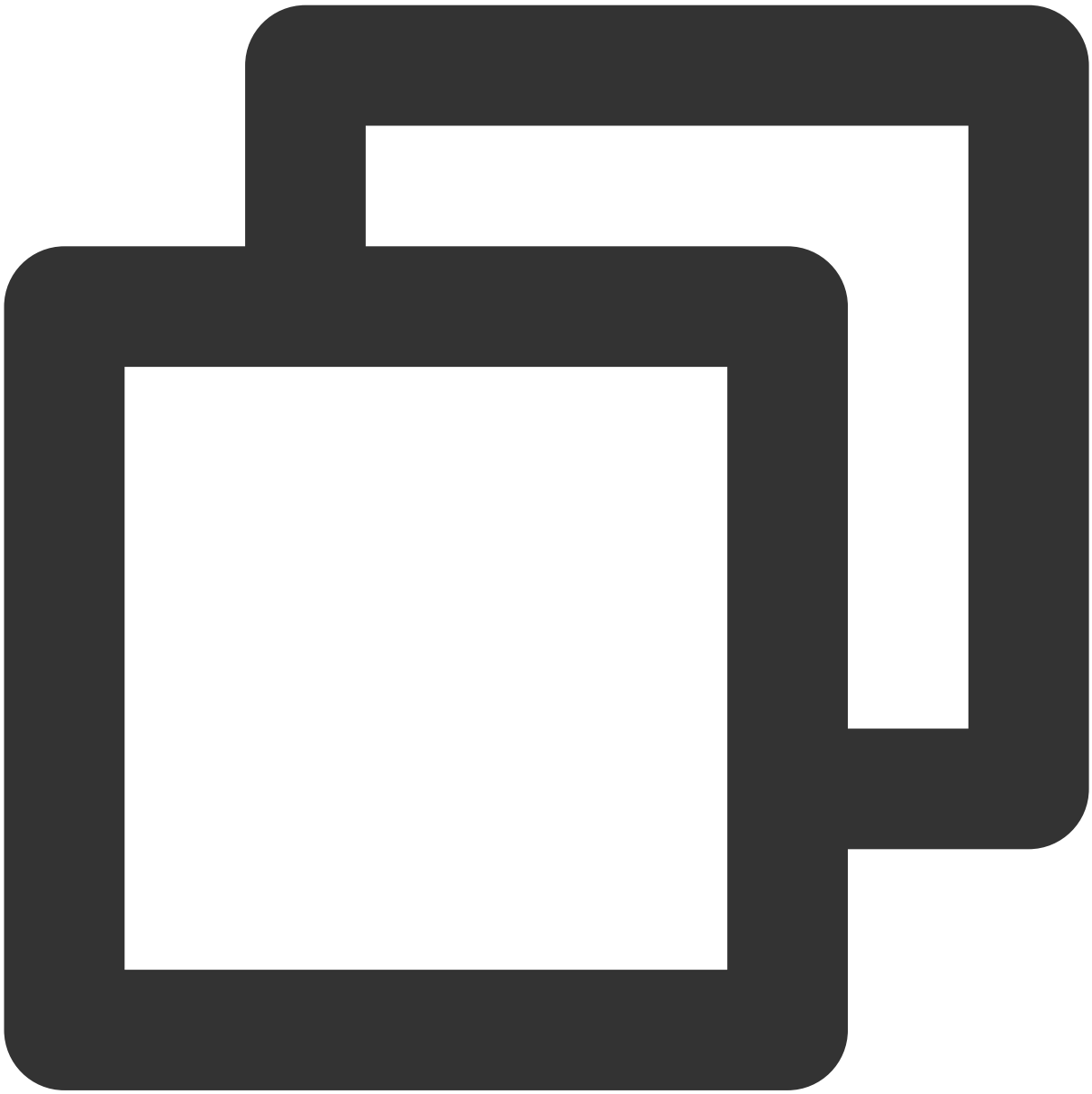
Server

1. Import the instrumentation dependency `opentracing-contrib/go-gin` on the server.

Dependency path: `github.com/opentracing-contrib/go-gin`

Version requirement: ≥ `v0.0.0-20201220185307-1dd2273433a4`

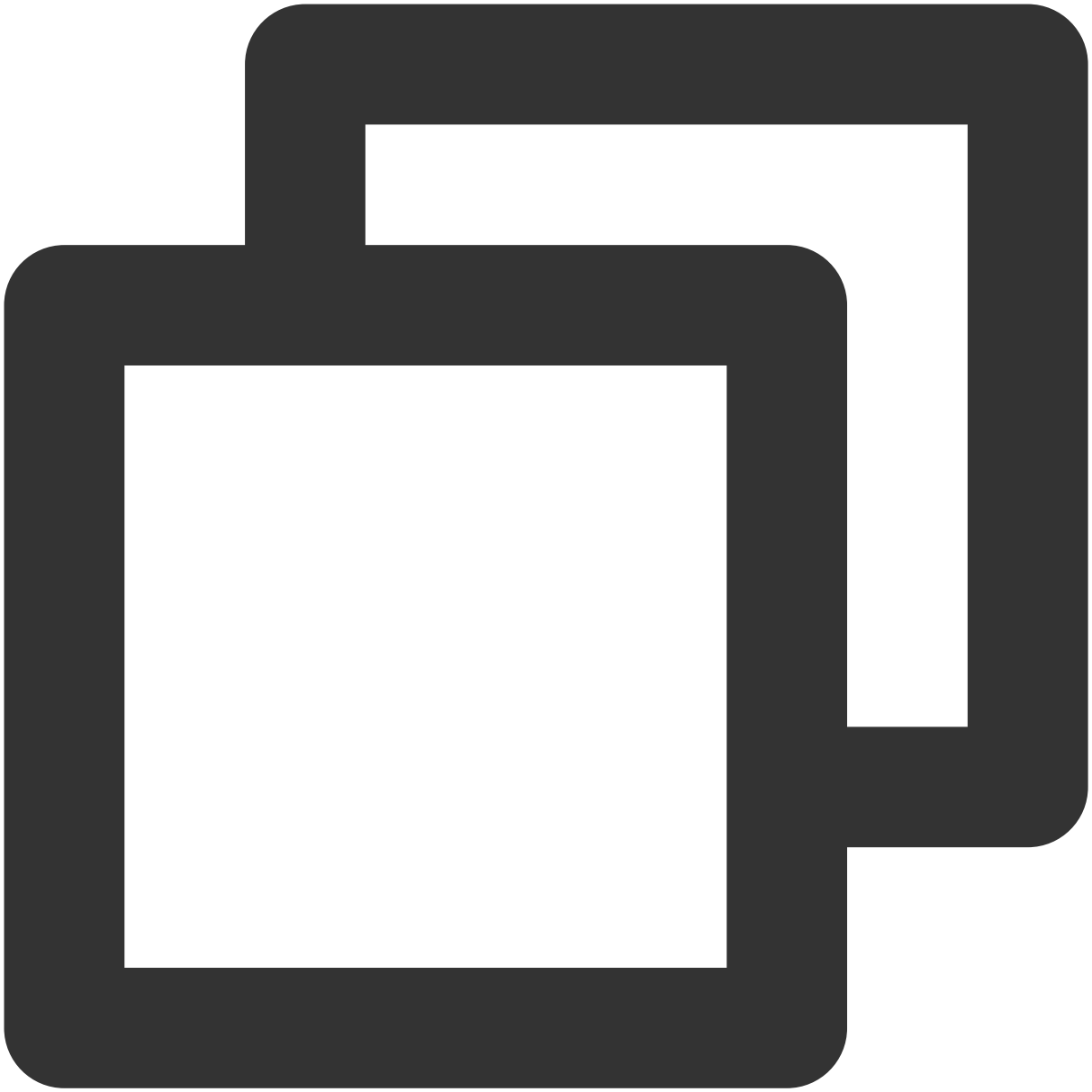
2. Configure Jaeger and create a trace object. Below is a sample:



```
cfg := &jaegerConfig.Configuration{
    ServiceName: ginServerName, // Call trace of the target service. Enter the serv
    Sampler: &jaegerConfig.SamplerConfig{ // Sampling policy configuration. See sec
    Type:  "const",
    Param: 1,
    },
    Reporter: &jaegerConfig.ReporterConfig{ // Configure how the client reports tra
```

```
      LogSpans:          true,
      LocalAgentHostPort: endPoint,
     },
     // Token configuration
     Tags:        []opentracing.Tag{ // Set the tag, where information such as token
     opentracing.Tag{Key: "token", Value: token}, // Set the token
     },
 }

 tracer, closer, err := cfg.NewTracer(jaegerConfig.Logger(jaeger.StdLogger)) // Get
```

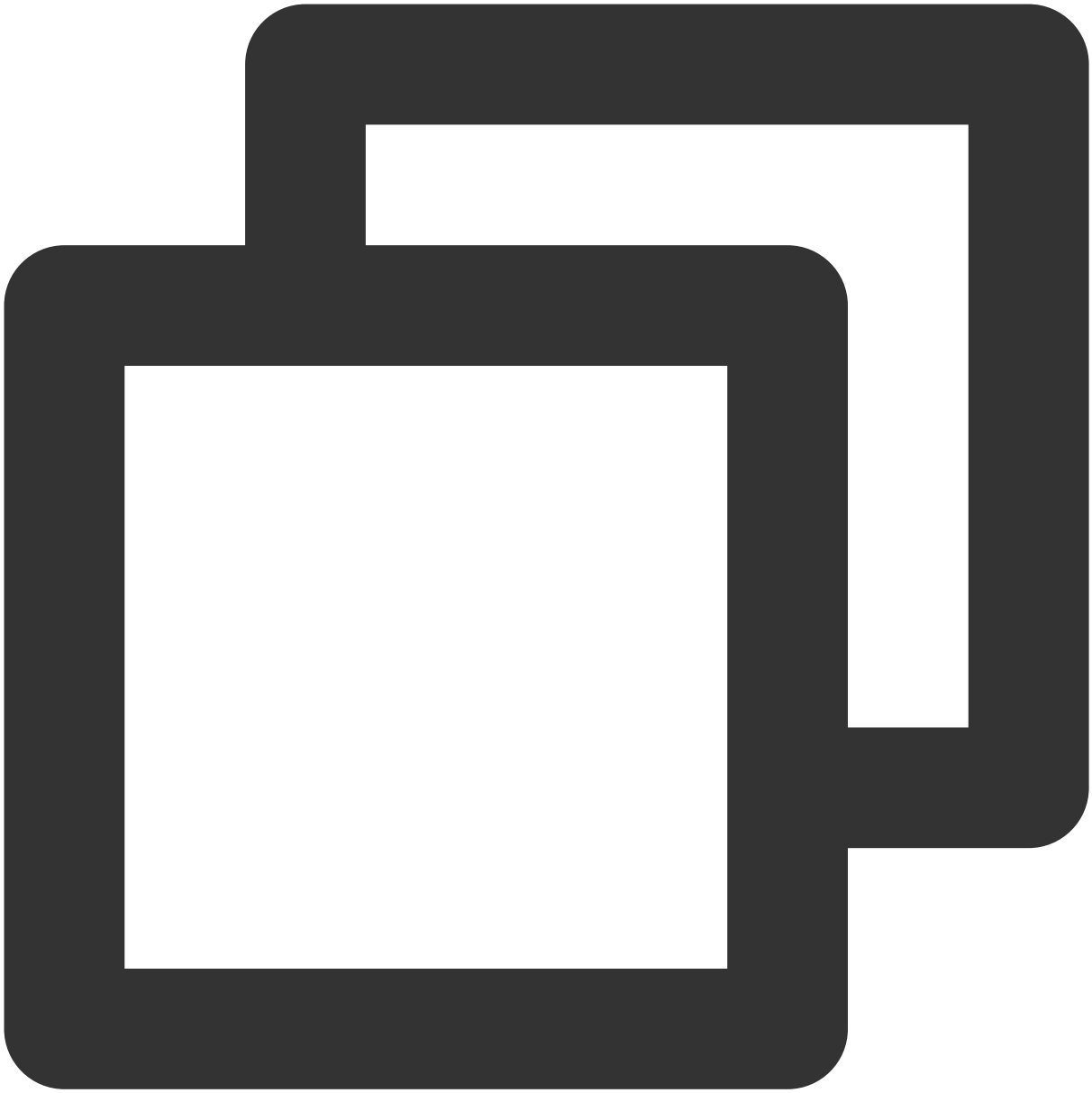
3. Configure the middleware.



```
r := gin.Default()
// Pass in the tracer
r.Use(ginhttp.Middleware(tracer))
```

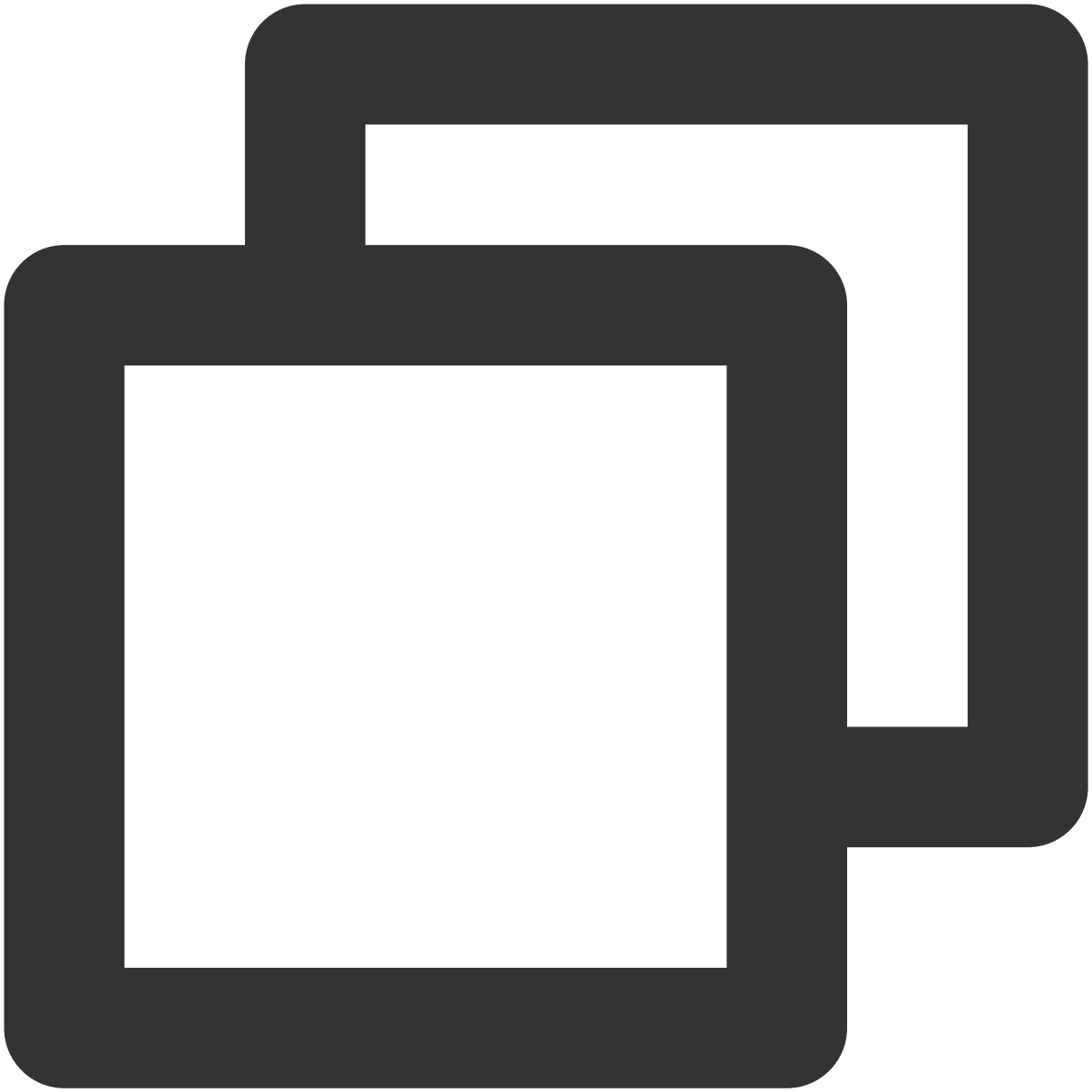
Note:

The official default OperationName is HTTP + HttpMethod. We recommend you use HTTP + HttpMethod + URL to analyze the specific API as follows. URL should be the parameter name rather than the specific parameter value. For example, `/user/{id}` is correct, while `/user/1` is incorrect.



```
r.Use(ginhttp.Middleware(tracer, ginhttp.OperationNameFunc(func(r *http.Request) st
        return fmt.Sprintf("testtestheling  HTTP %s %s", r.Method, r.URL.String())
    })))
```

The complete code is as follows:



```
// Copyright © 2019-2020 Tencent Co., Ltd.

// This file is part of tencent project.
// Do not copy, cite, or distribute without the express
// permission from Cloud Monitor group.

package gindemo

import (
    "fmt"
    "github.com/gin-gonic/gin"
```

```
    "github.com/opentracing-contrib/go-gin/ginhttp"
    "github.com/opentracing/opentracing-go"
    "github.com/uber/jaeger-client-go"
    jaegerConfig "github.com/uber/jaeger-client-go/config"
    "net/http"
)

// Service name, which is the unique identifier of the service and the basis for se
const ginServerName = "demo-gin-server"

// StartServer
func StartServer() {
    // Initialize Jaeger to get the tracer
    cfg := &jaegerConfig.Configuration{
        ServiceName: ginServerName, // Call trace of the target service. Enter the
        Sampler: &jaegerConfig.SamplerConfig{ // Sampling policy configuration. See
            Type:  "const",
            Param: 1,
        },
        Reporter: &jaegerConfig.ReporterConfig{ // Configure how the client reports
            LogSpans:          true,
            LocalAgentHostPort: endPoint,
        },
        // Token configuration
        Tags: []opentracing.Tag{ // Set the tag, where information such as token ca
            opentracing.Tag{Key: "token", Value: token}, // Set the token
        },
    }

    tracer, closer, err := cfg.NewTracer(jaegerConfig.Logger(jaeger.StdLogger)) //
    if err != nil {
        panic(fmt.Sprintf("ERROR: fail init Jaeger: %v\\n", err))
    }
    defer closer.Close()
    r := gin.Default()
    // Note that the official default OperationName is HTTP + HttpMethod.
    // We recommend you use HTTP + HttpMethod + URL to analyze the specific API as
    // Note: For RESTful APIs, URL should be the parameter name rather than the spe
    r.Use(ginhttp.Middleware(tracer, ginhttp.OperationNameFunc(func(r *http.Request
        return fmt.Sprintf("HTTP %s %s", r.Method, r.URL.String())
    })))
    r.GET("/ping", func(c *gin.Context) {
        c.JSON(200, gin.H{
            "message": "pong",
        })
    })
    r.Run() // Listen on 0.0.0.0:8080
```

```
}
```

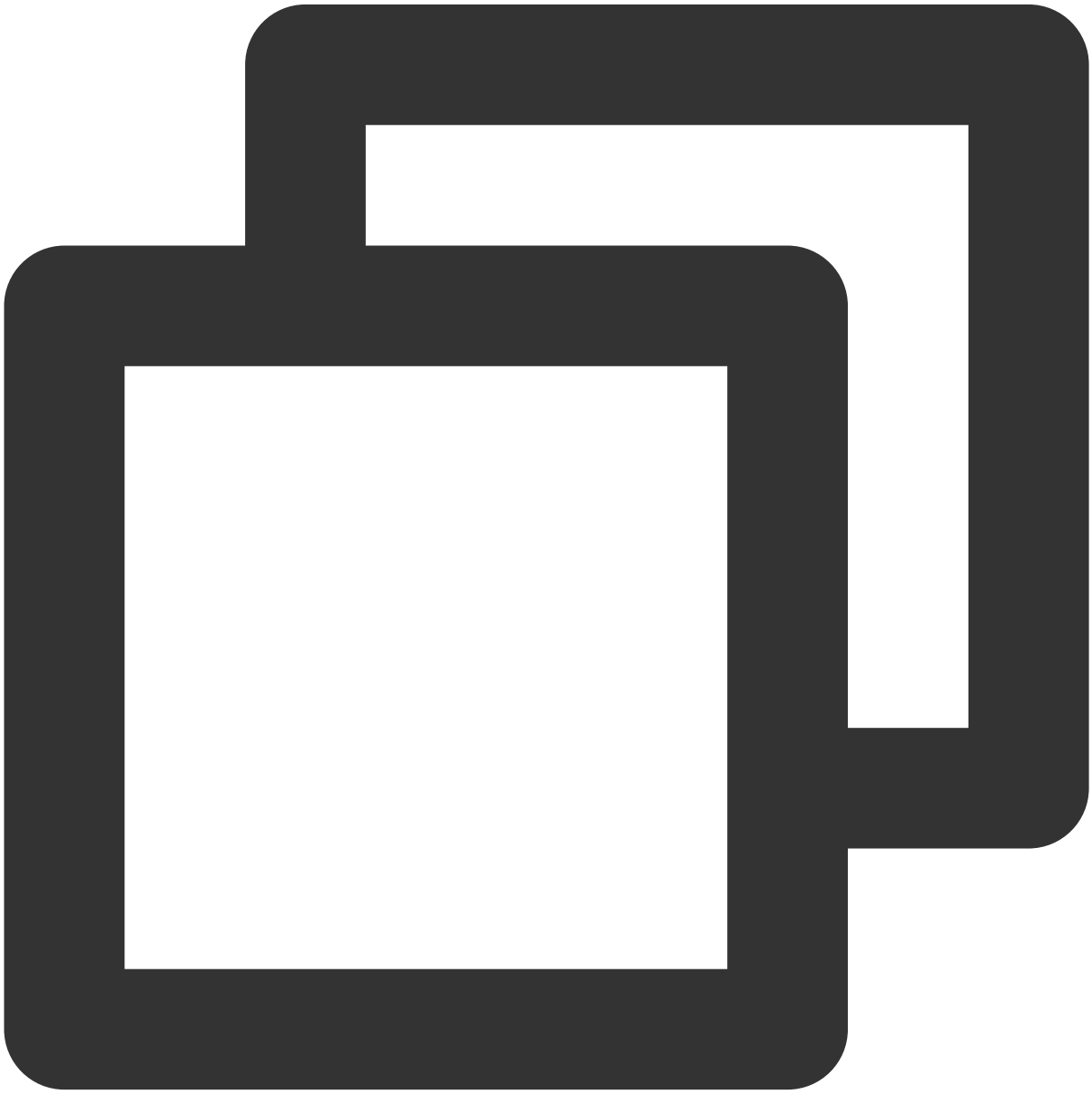
Client

1. Due to the need to simulate HTTP requests on the client, import the `opentracing-contrib/go-stdlib/nethttp` dependency.

Dependency path: `github.com/opentracing-contrib/go-stdlib/nethttp`

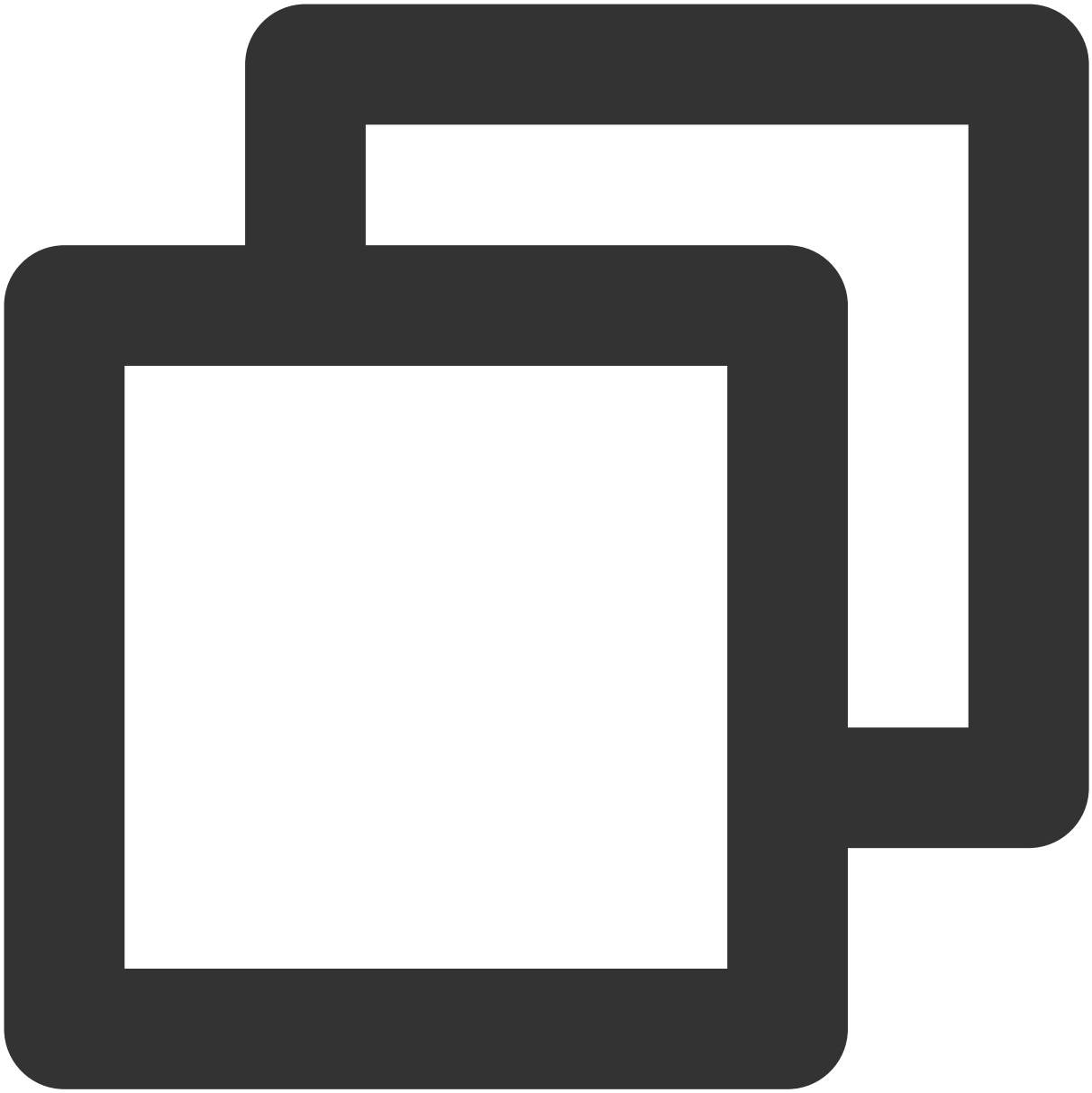
Version requirement: ≥ `v1.0.0`

2. Configure Jaeger and create a trace object. Below is a sample:



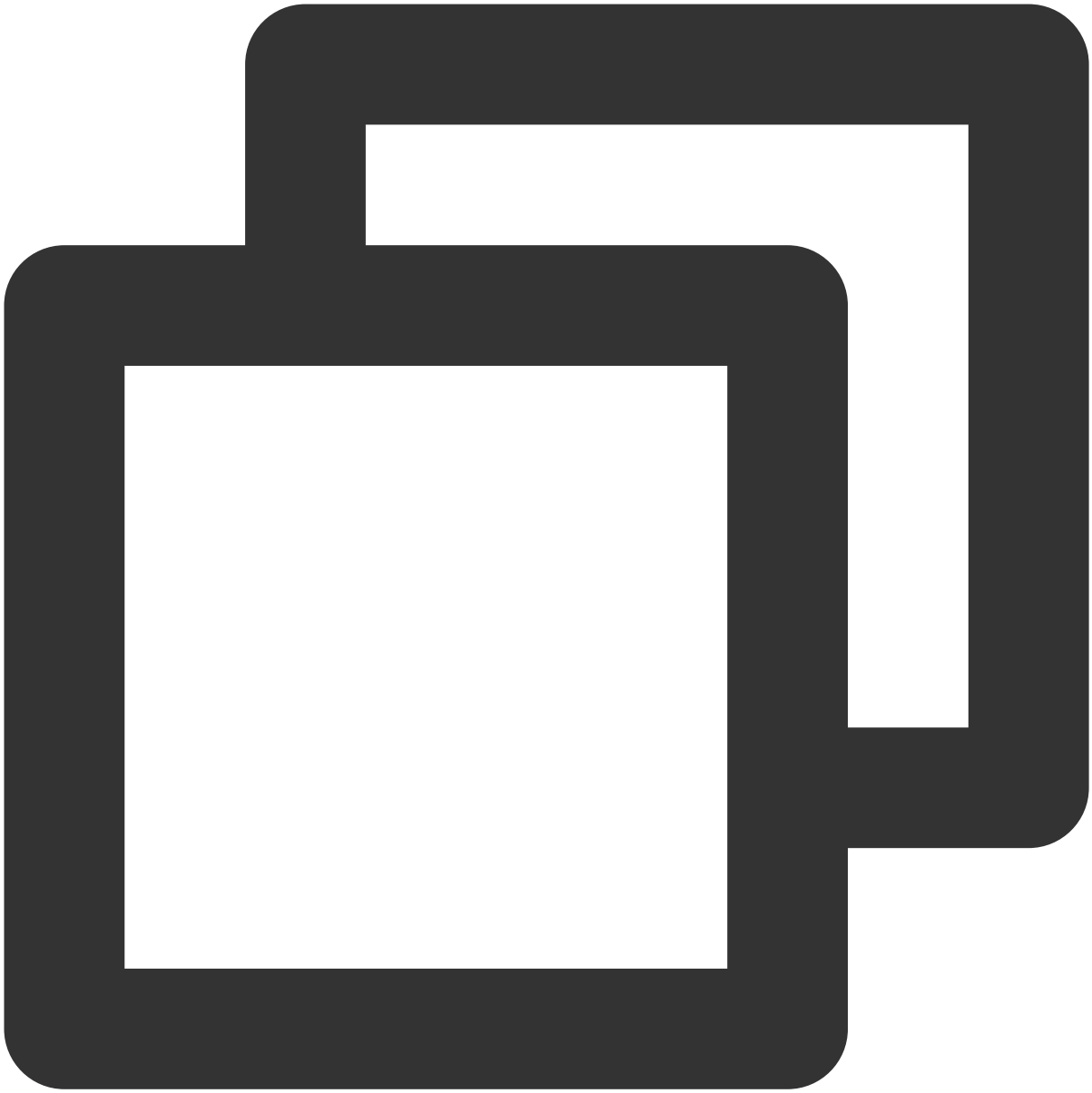
```
cfg := &jaegerConfig.Configuration{
    ServiceName: ginClientName, // Call trace of the target service. Enter the serv
    Sampler: &jaegerConfig.SamplerConfig{ // Sampling policy configuration. See sec
    Type:  "const",
    Param: 1,
    },
    Reporter: &jaegerConfig.ReporterConfig{ // Configure how the client reports tra
    LogSpans:          true,
    LocalAgentHostPort: endPoint,
    },
    // Token configuration
    Tags:        []opentracing.Tag{ // Set the tag, where information such as token
    opentracing.Tag{Key: "token", Value: token}, // Set the token
    },
}

tracer, closer, err := cfg.NewTracer(jaegerConfig.Logger(jaeger.StdLogger)) // Get
```

3. Construct a span and put it in the context. Below is a sample:



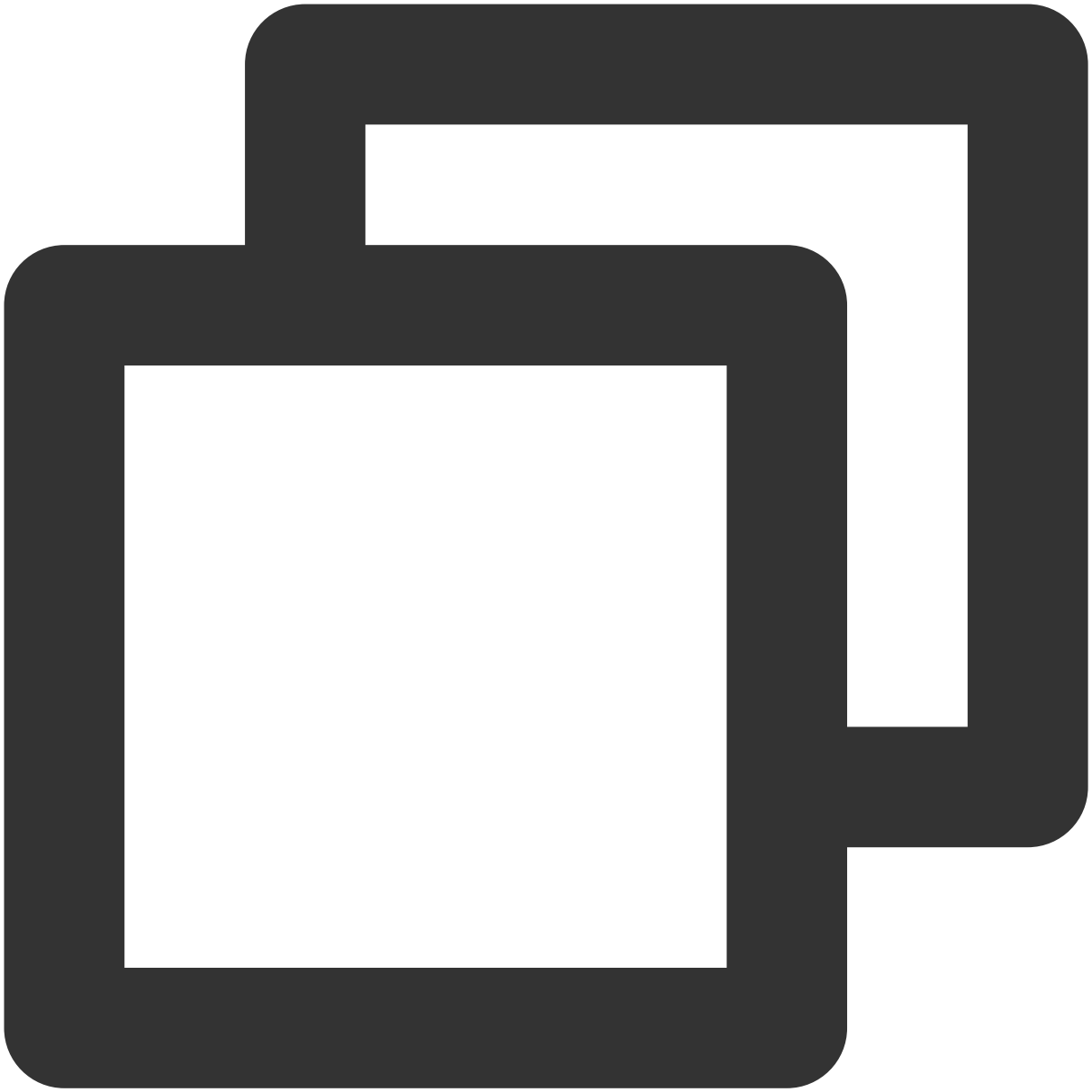
```
span := tracer.StartSpan("CallDemoServer") // Construct a span
ctx := opentracing.ContextWithSpan(context.Background(), span) // Put the span refe
```

4. Construct a request with the tracer. Below is a sample:



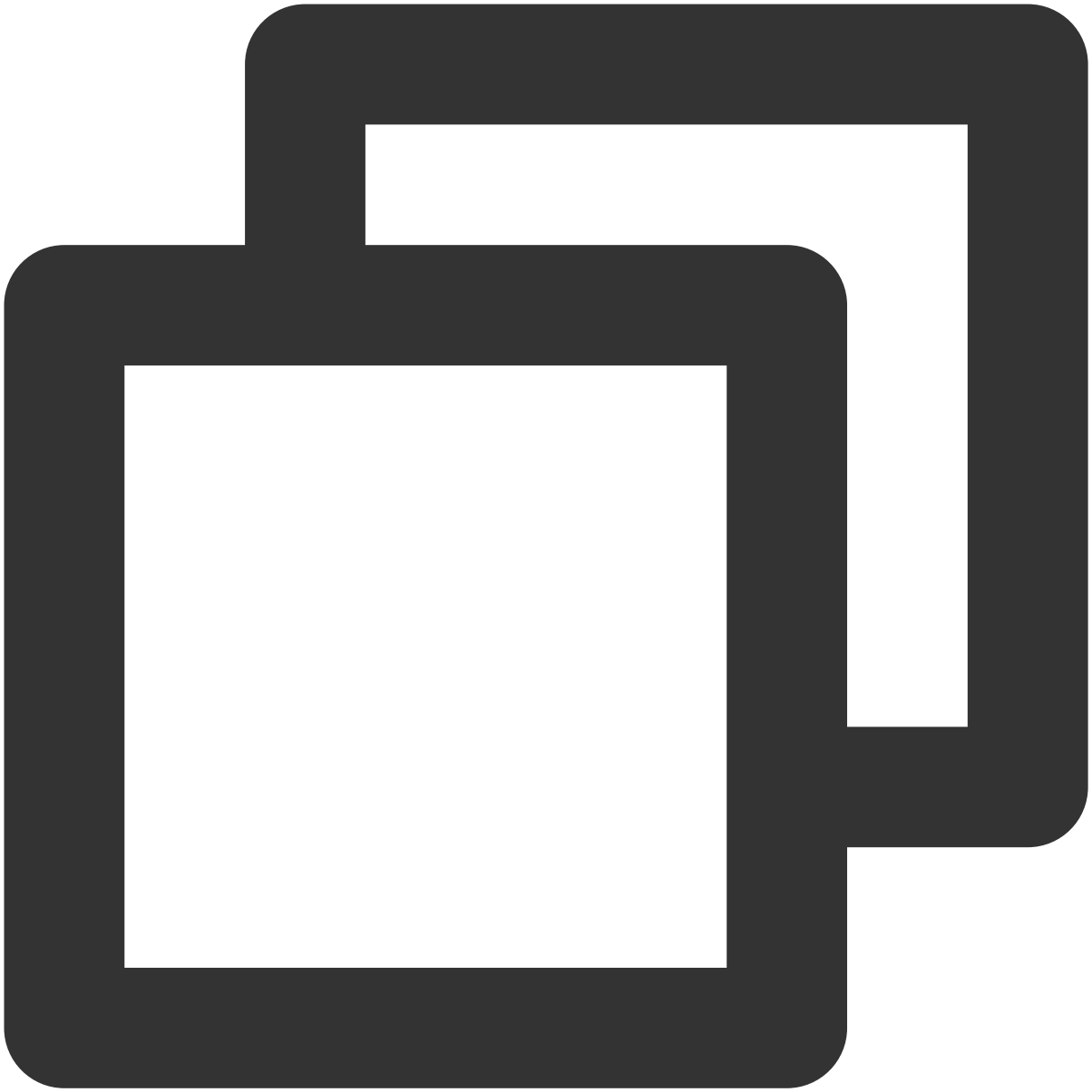
```
// Construct an HTTP request
req, err := http.NewRequest(
        http.MethodGet,
        fmt.Sprintf("http://localhost%s/ping", ginPort),
        nil,
    )
req = req.WithContext(ctx)
// Construct a request with the tracer
req, ht := nethttp.TraceRequest(tracer, req)
```

5. Send an HTTP request and get the returned result.



```
httpClient := &http.Client{Transport: &nethttp.Transport{}} // Initialize the HTTP
res, err := httpClient.Do(req)
// ... Error judgment is omitted.
body, err := ioutil.ReadAll(res.Body)
// ... Error judgment is omitted.
log.Printf(" %s recevice: %s\\n", clientServerName, string(body))
```

The complete code is as follows:



```
// Copyright © 2019-2020 Tencent Co., Ltd.

// This file is part of tencent project.
// Do not copy, cite, or distribute without the express
// permission from Cloud Monitor group.

package gindemo

import (
    "context"
    "fmt"
```

```
    "github.com/opentracing-contrib/go-stdlib/nethttp"
    "github.com/opentracing/opentracing-go"
    "github.com/opentracing/opentracing-go/ext"
    opentracingLog "github.com/opentracing/opentracing-go/log"
    "github.com/uber/jaeger-client-go"
    jaegerConfig "github.com/uber/jaeger-client-go/config"
    "io/ioutil"
    "log"
    "net/http"
)

const (
    // Service name, which is the unique identifier of the service and the basis fo
    ginClientName = "demo-gin-client"
    ginPort       = ":8080"
    endPoint      = "xxxxx:6831" // Local agent address
    token         = "abc"
)

// The Gin client under StartClient is also a standard HTTP client.
func StartClient() {
    cfg := &jaegerConfig.Configuration{
        ServiceName: ginClientName, // Call trace of the target service. Enter the
        Sampler: &jaegerConfig.SamplerConfig{ // Sampling policy configuration. See
            Type:  "const",
            Param: 1,
        },
        Reporter: &jaegerConfig.ReporterConfig{ // Configure how the client reports
            LogSpans:           true,
            LocalAgentHostPort: endPoint,
        },
        // Token configuration
        Tags: []opentracing.Tag{ // Set the tag, where information such as token ca
            opentracing.Tag{Key: "token", Value: token}, // Set the token
        },
    }

    tracer, closer, err := cfg.NewTracer(jaegerConfig.Logger(jaeger.StdLogger)) //
    defer closer.Close()
    if err != nil {
        panic(fmt.Sprintf("ERROR: fail init Jaeger: %v\\n", err))
    }
    // Construct a span and put it in the context
    span := tracer.StartSpan("CallDemoServer")
    ctx := opentracing.ContextWithSpan(context.Background(), span)
    defer span.Finish()
```

```
    // Construct an HTTP request
    req, err := http.NewRequest(
        http.MethodGet,
        fmt.Sprintf("http://localhost%s/ping", ginPort),
        nil,
    )
    if err != nil {
        HandlerError(span, err)
        return
    }
    // Construct a request with a tracer
    req = req.WithContext(ctx)
    req, ht := nethttp.TraceRequest(tracer, req)
    defer ht.Finish()

    // Initialize the HTTP client
    httpClient := &http.Client{Transport: &nethttp.Transport{}}
    // Send the request
    res, err := httpClient.Do(req)
    if err != nil {
        HandlerError(span, err)
        return
    }
    defer res.Body.Close()
    body, err := ioutil.ReadAll(res.Body)
    if err != nil {
        HandlerError(span, err)
        return
    }
    log.Printf(" %s recevice: %s\\n", ginClientName, string(body))
}

// HandlerError handle error to span.
func HandlerError(span opentracing.Span, err error) {
    span.SetTag(string(ext.Error), true)
    span.LogKV(opentracingLog.Error(err))
}
```

Reporting with go-redis Middleware

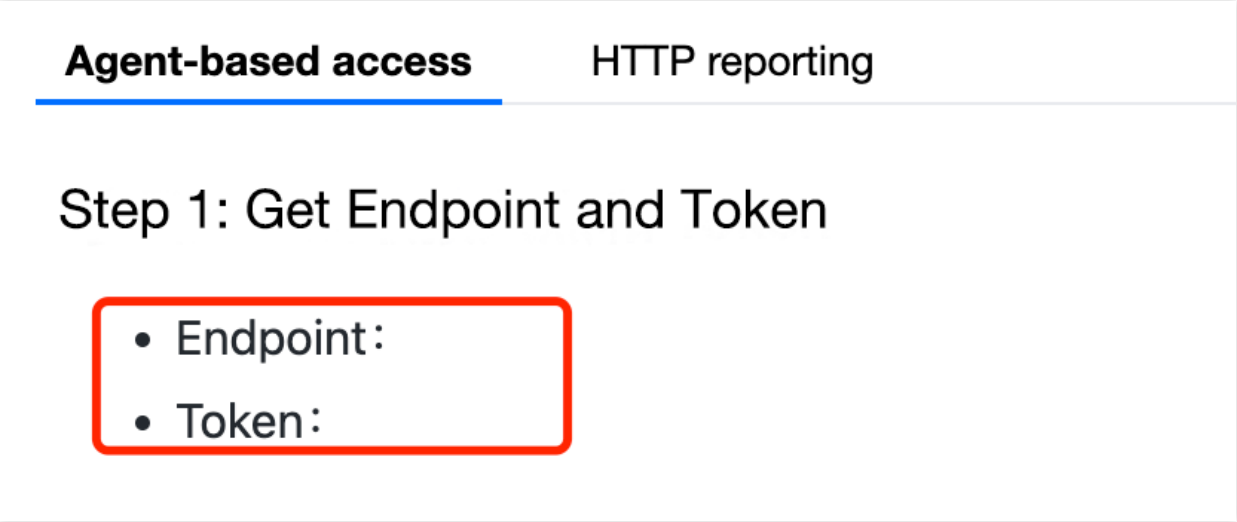
Last updated：2023-12-25 16:00:35

This document describes how to report the data of a Go application with the go-redis middleware.

Directions

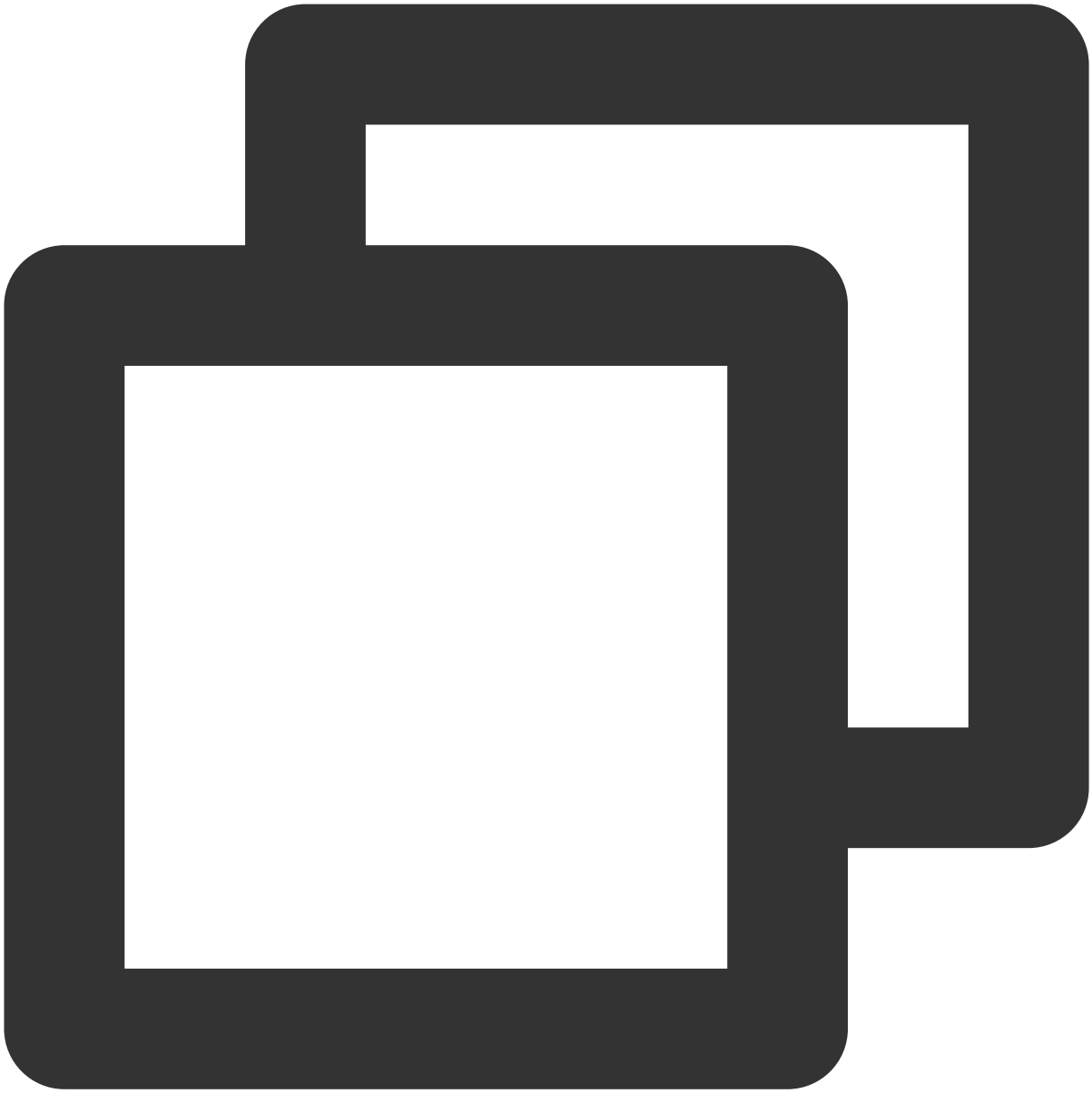
Step 1. Get the endpoint and token

Log in to the APM console, enter the **Application monitoring** > **Application list** page, click **Access application**, and select the Go language and the go-redis data collection method.
Then, get the endpoint and token in the step of access method selection.



Step 2. Install the Jaeger agent

1. Download the Jaeger agent.
2. Run the following command to start the agent:



```
nohup ./jaeger-agent --reporter.grpc.host-port={{collectorRPCHostPort}} --agent.tag
```

Step 3. Select the reporting type to report application data

Select the reporting type to report the data of a Go application through the go-redis middleware:

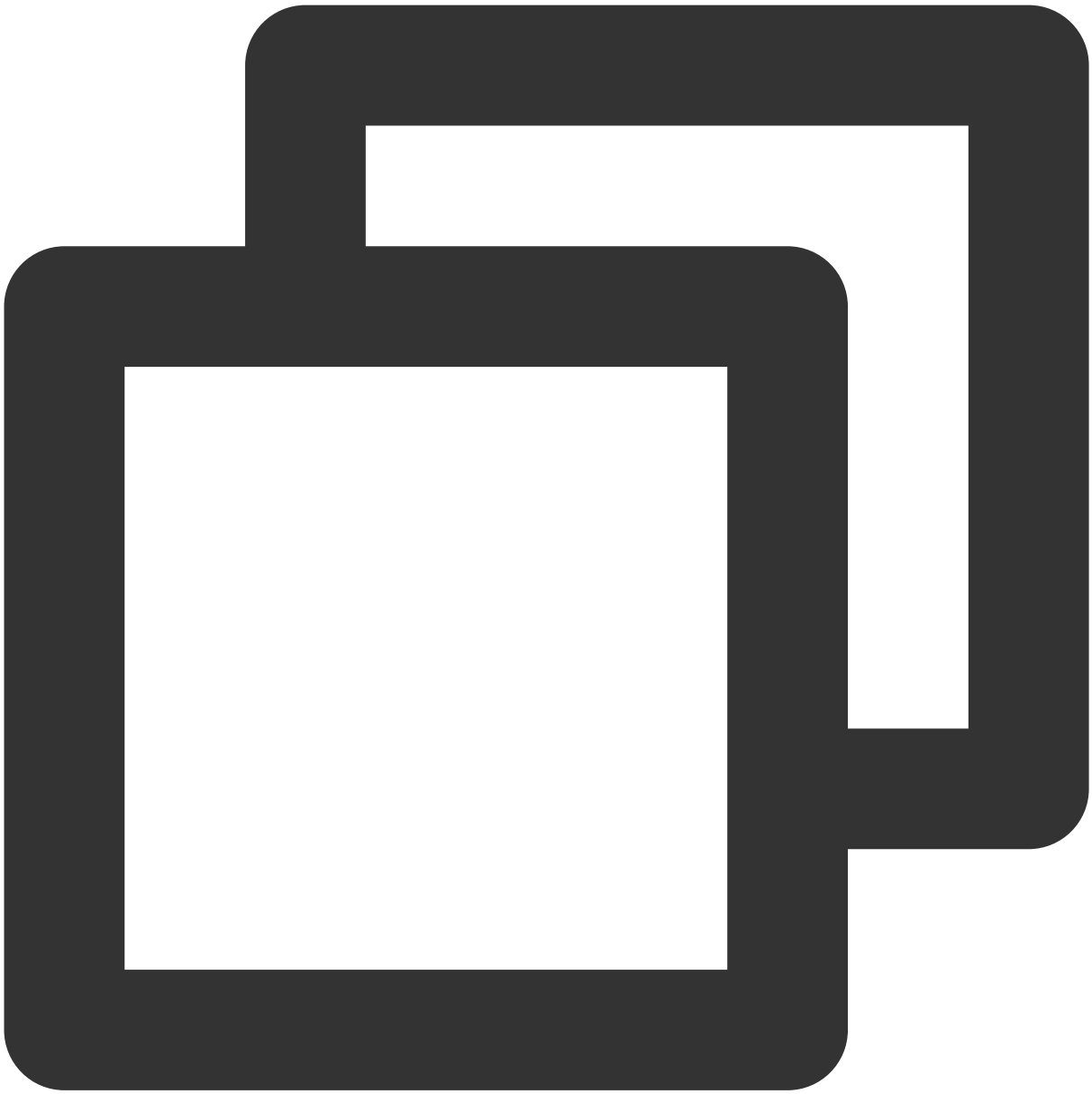
Client

1. Import the instrumentation dependency `opentracing-contrib/goredis` .

Dependency path: `github.com/opentracing-contrib/goredis`

Version requirement: ≥ `v0.0.0-20190807091203-90a2649c5f87`

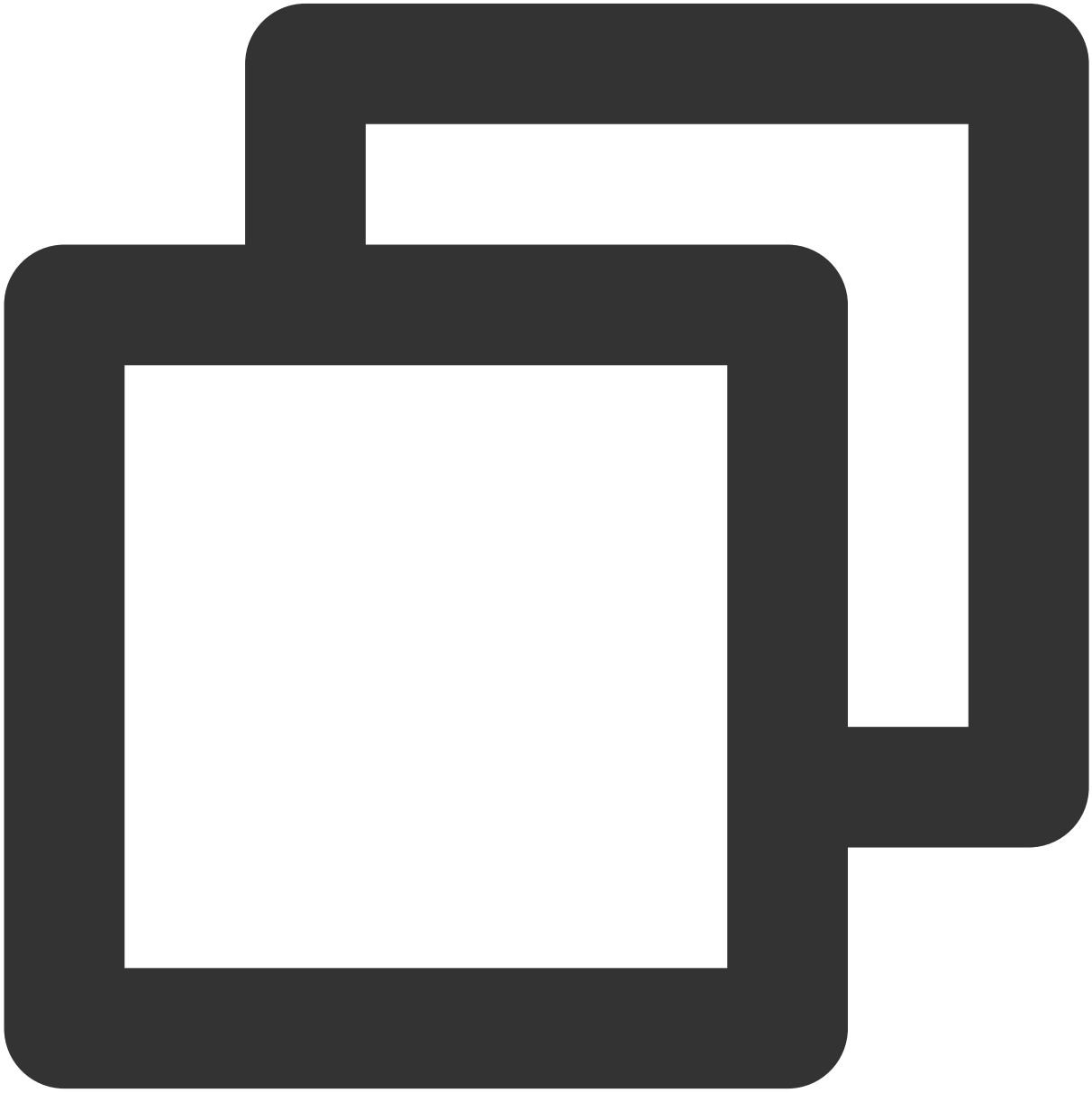
2. Configure Jaeger, create a trace object, and set GlobalTracer. Below is a sample:



```
cfg := &jaegerConfig.Configuration{
    ServiceName: clientServerName, // Call trace of the target service. Enter the s
    Sampler: &jaegerConfig.SamplerConfig{ // Sampling policy configuration. See sec
    Type:  "const",
    Param: 1,
    },
    Reporter: &jaegerConfig.ReporterConfig{ // Configure how the client reports tra
    LogSpans:          true,
```

```
        LocalAgentHostPort: endPoint,
    },
    // Token configuration
    Tags:        []opentracing.Tag{ // Set the tag, where information such as token
    opentracing.Tag{Key: "token", Value: token}, // Set the token
    },
}
tracer, closer, err := cfg.NewTracer(jaegerConfig.Logger(jaeger.StdLogger)) // Get
```

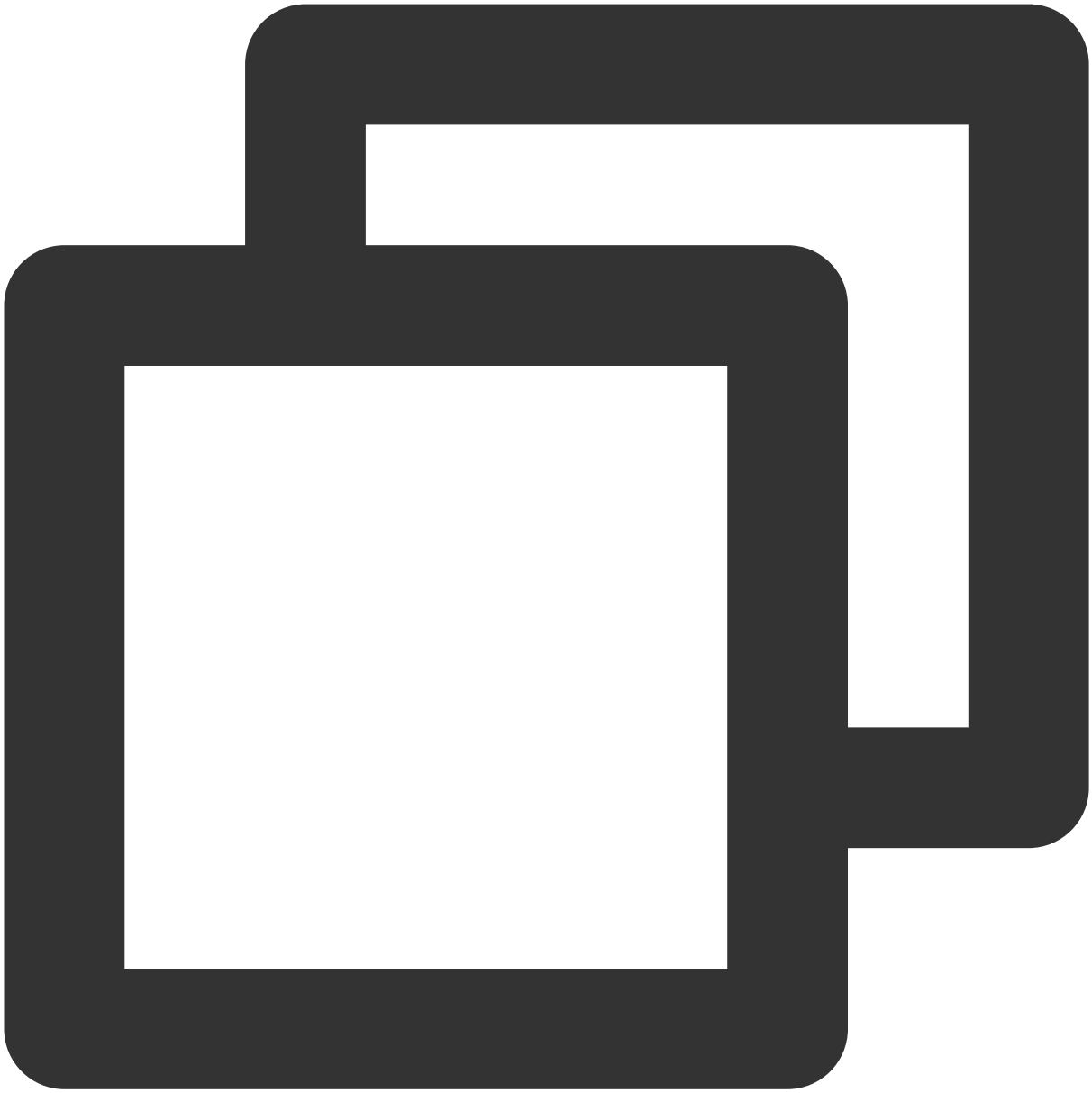
3. Initialize the Redis connection. Below is a sample:



```
func InitRedisConnector() error {
```

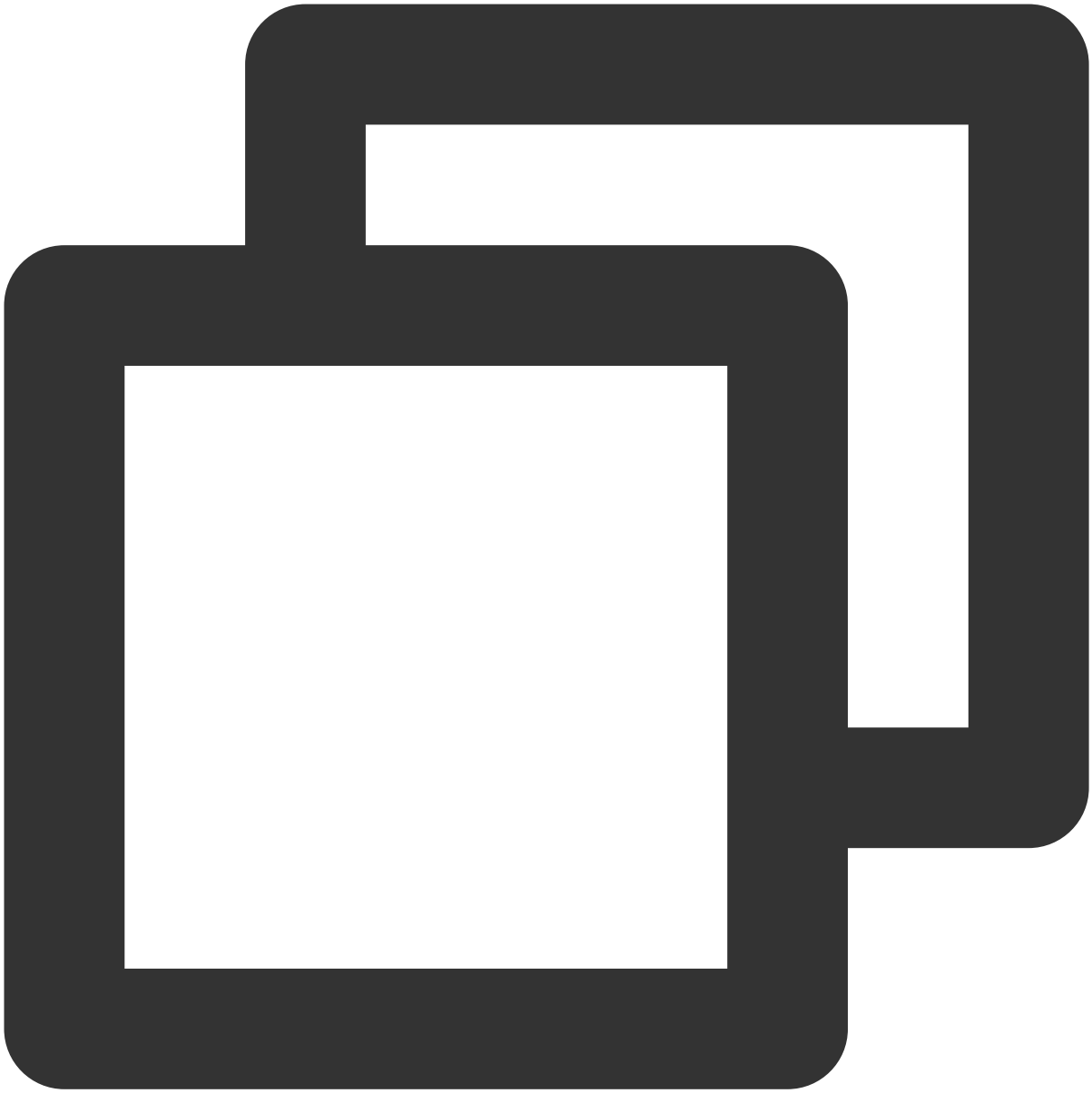
```
    redisClient = redis.NewUniversalClient(&redis.UniversalOptions{
        Addrs:    []string{redisAddress},
        Password: redisPassword,
        DB:       0,
    })
    if err := redisClient.Ping().Err(); err != nil {
        log.Println("redisClient.Ping() error:", err.Error())
        return err
    }
    return nil
}
```

4. Get the Redis connection. Below is a sample:



```
func GetRedisDBConnector(ctx context.Context) redis.UniversalClient {
    client := apmgoredis.Wrap(redisClient).WithContext(ctx)
    return client
}
```

The complete code is as follows:



```
package main

import (
    "context"
    "fmt"
    "github.com/go-redis/redis"
    apmgoredis "github.com/opentracing-contrib/goredis"
    "github.com/opentracing/opentracing-go"
    "github.com/uber/jaeger-client-go"
    jaegerConfig "github.com/uber/jaeger-client-go/config"
    "log"
```

```
    "time"
)

const (
    redisAddress     = "127.0.0.1:6379"
    redisPassword    = ""
    clientServerName = "redis-client-demo"
    testKey          = "redis-demo-key"
    endPoint         = "xxxxx:6831" // HTTP reporting address
    token            = "abc"
)

func main() {
    cfg := &jaegerConfig.Configuration{
        ServiceName: clientServerName, // Call trace of the target service. Enter t
        Sampler: &jaegerConfig.SamplerConfig{ // Sampling policy configuration. See
            Type:  "const",
            Param: 1,
        },
        Reporter: &jaegerConfig.ReporterConfig{ // Configure how the client reports
            LogSpans:           true,
            LocalAgentHostPort: endPoint,
        },
        // Token configuration
        Tags: []opentracing.Tag{ // Set the tag, where information such as token ca
            opentracing.Tag{Key: "token", Value: token}, // Set the token
        },
    }
    tracer, closer, err := cfg.NewTracer(jaegerConfig.Logger(jaeger.StdLogger)) //
    opentracing.SetGlobalTracer(tracer)
    defer closer.Close()
    if err != nil {
        panic(fmt.Sprintf("ERROR: fail init Jaeger: %v\\n", err))
    }
    InitRedisConnector()
    redisClient := GetRedisDBConnector(context.Background())
    redisClient.Set(testKey, "redis-client-demo", time.Duration(1000)*time.Second)
    redisClient.Get(testKey)
}

var (
    redisClient redis.UniversalClient
)

func GetRedisDBConnector(ctx context.Context) redis.UniversalClient {
    client := apmgoredis.Wrap(redisClient).WithContext(ctx)
    return client
```

```
}
func InitRedisConnector() error {
    redisClient = redis.NewUniversalClient(&redis.UniversalOptions{
        Addrs:    []string{redisAddress},
        Password: redisPassword,
        DB:       0,
    })
    if err := redisClient.Ping().Err(); err != nil {
        log.Println("redisClient.Ping() error:", err.Error())
        return err
    }
    return nil
}
```

Reporting with gRPC-Jaeger Interceptor

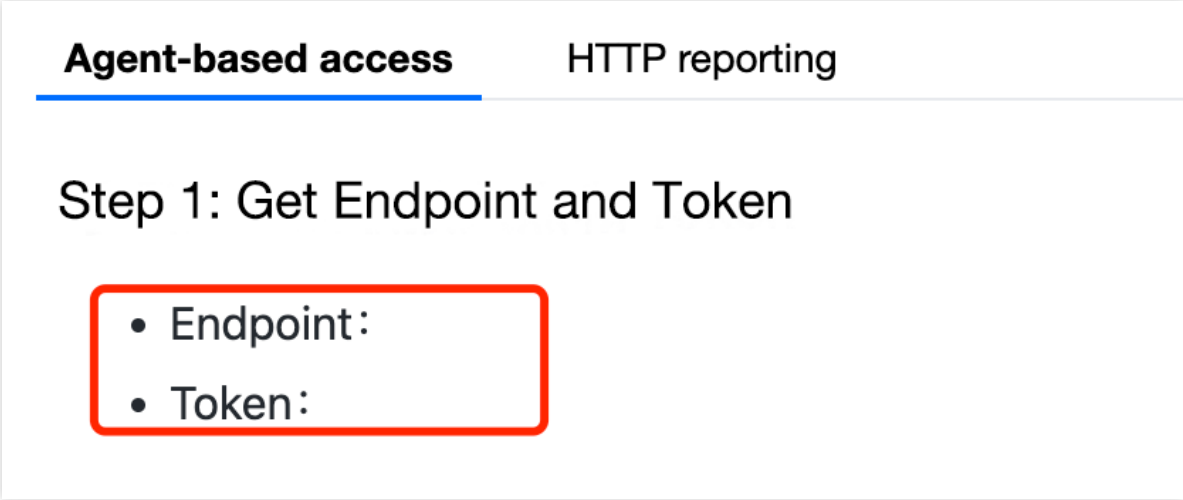
Last updated：2023-12-25 16:00:53

This document describes how to report the data of a Go application with the gRPC-Jaeger interceptor.

Directions

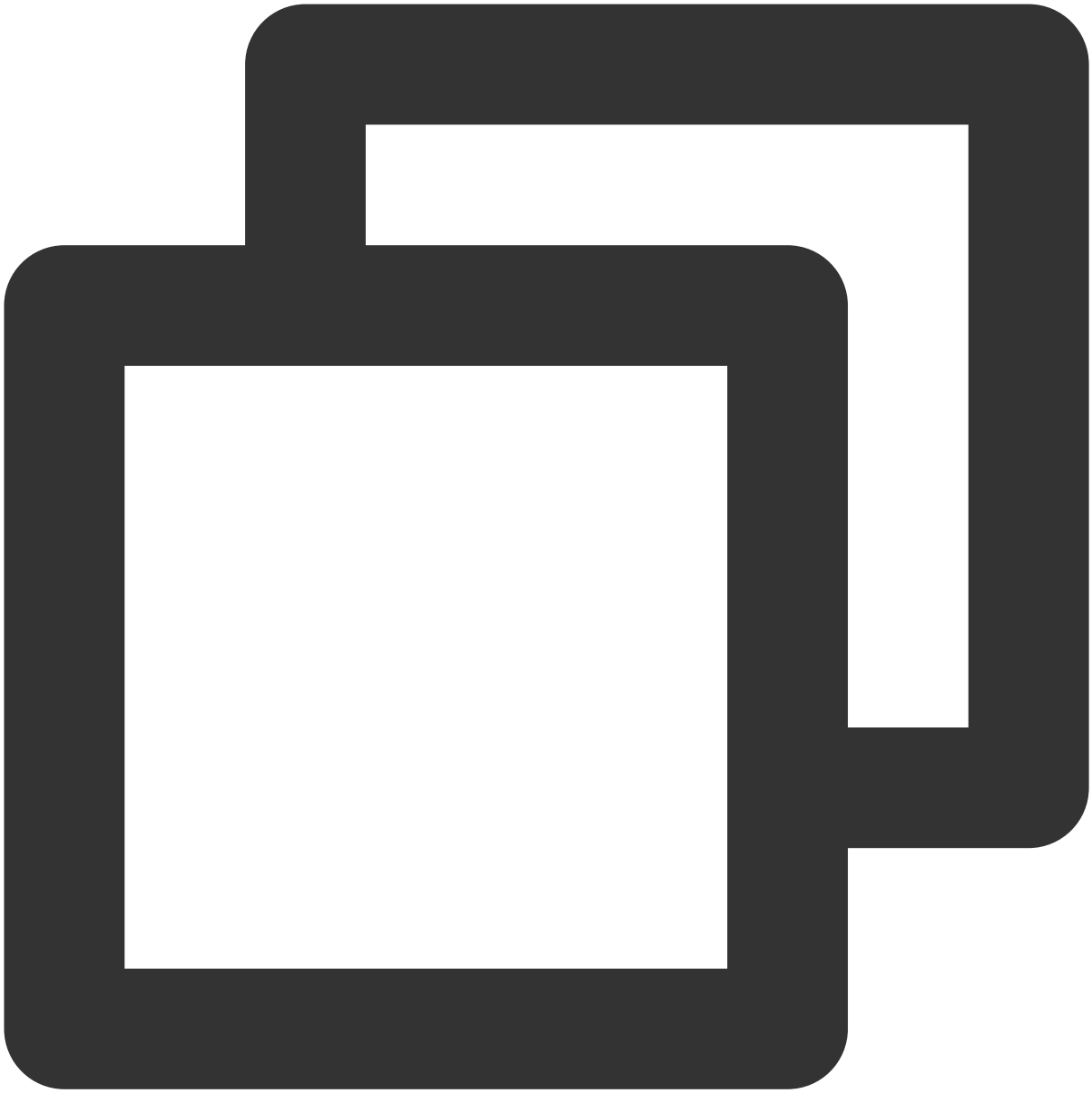
Step 1. Get the endpoint and token

Log in to the APM console, enter the **Application monitoring** > **Application list** page, click **Access application**, and select the Go language and the gRPC-Jaeger data collection method.
Then, get the endpoint and token in the step of access method selection.



Step 2. Install the Jaeger agent

1. Download the Jaeger agent.
2. Run the following command to start the agent:



```
nohup ./jaeger-agent --reporter.grpc.host-port={{collectorRPCHostPort}} --agent.tag
```

Step 3. Select the reporting type to report application data

Select the reporting type to report the data of a Go application through the gRPC-Jaeger interceptor:

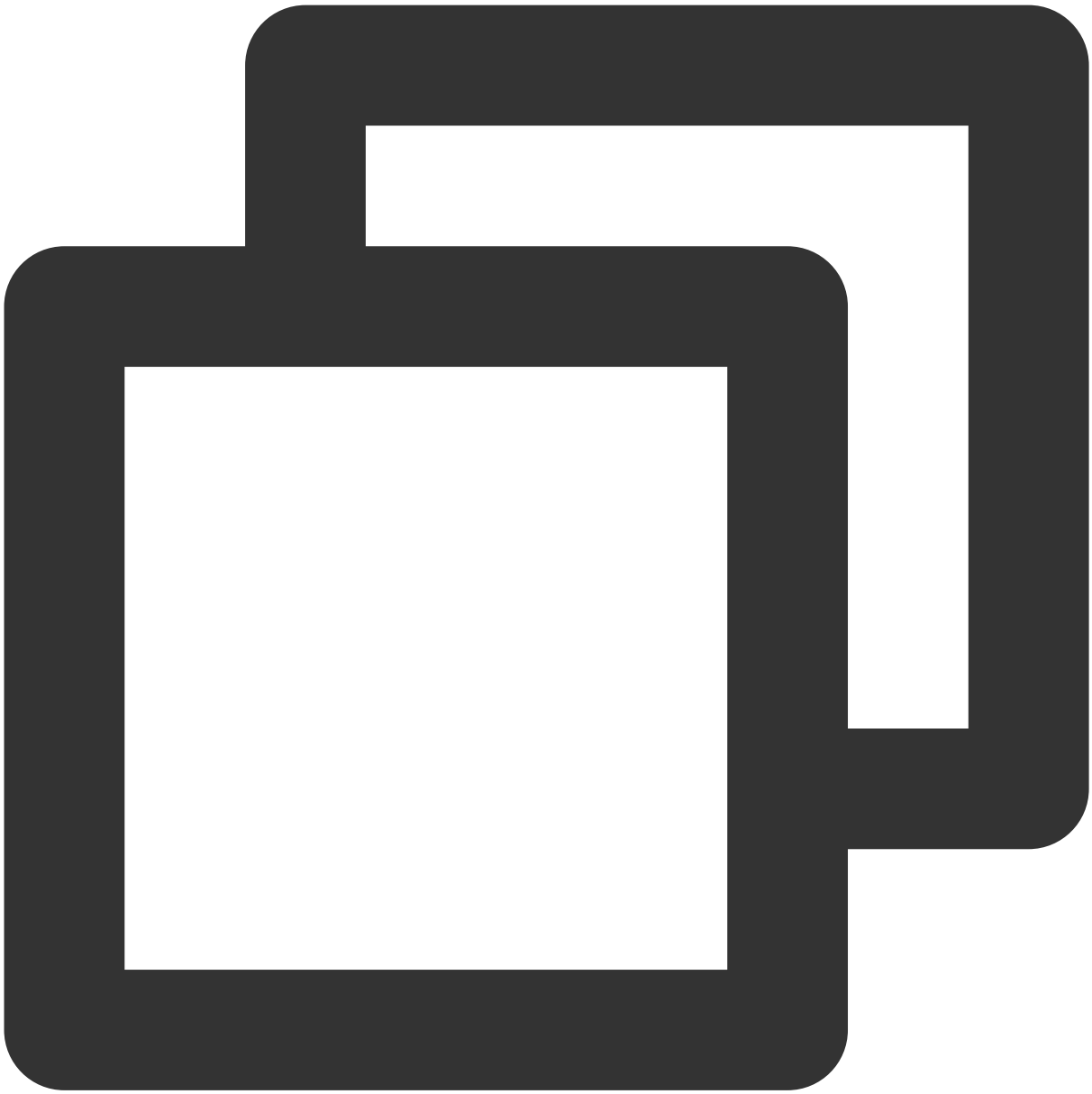
Server

1. Import the instrumentation dependency `opentracing-contrib/go-grpc` on the server.

Dependency path: `github.com/opentracing-contrib/go-grpc`

Version requirement: ≥ `v0.0.0-20210225150812-73cb765af46e`

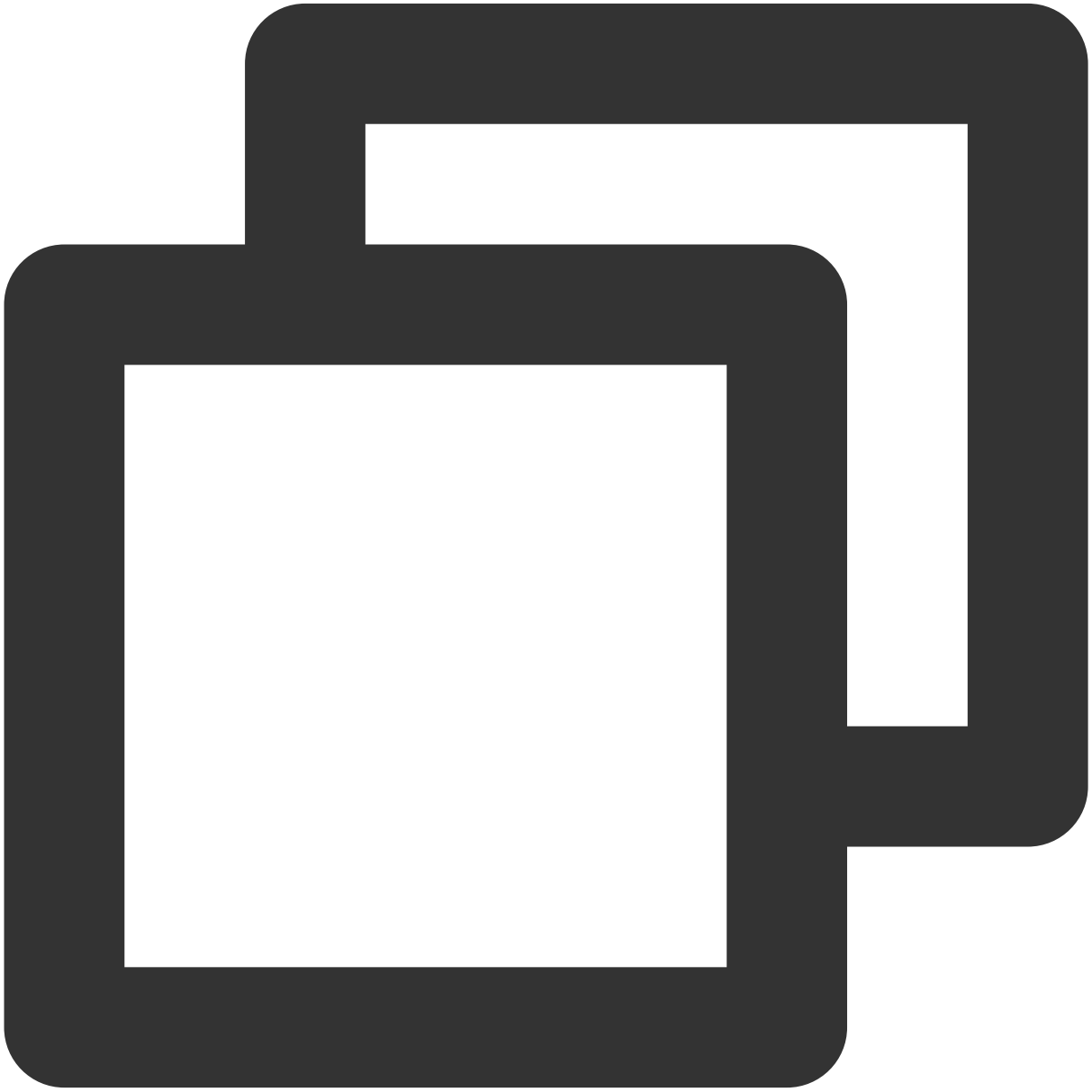
2. Configure Jaeger and create a trace object. Below is a sample:



```
cfg := &jaegerConfig.Configuration{
    ServiceName: grpcServerName, // Call trace of the target service. Enter the ser
    Sampler: &jaegerConfig.SamplerConfig{ // Sampling policy configuration. See sec
    Type:  "const",
    Param: 1,
    },
    Reporter: &jaegerConfig.ReporterConfig{ // Configure how the client reports tra
    LogSpans:          true,
```

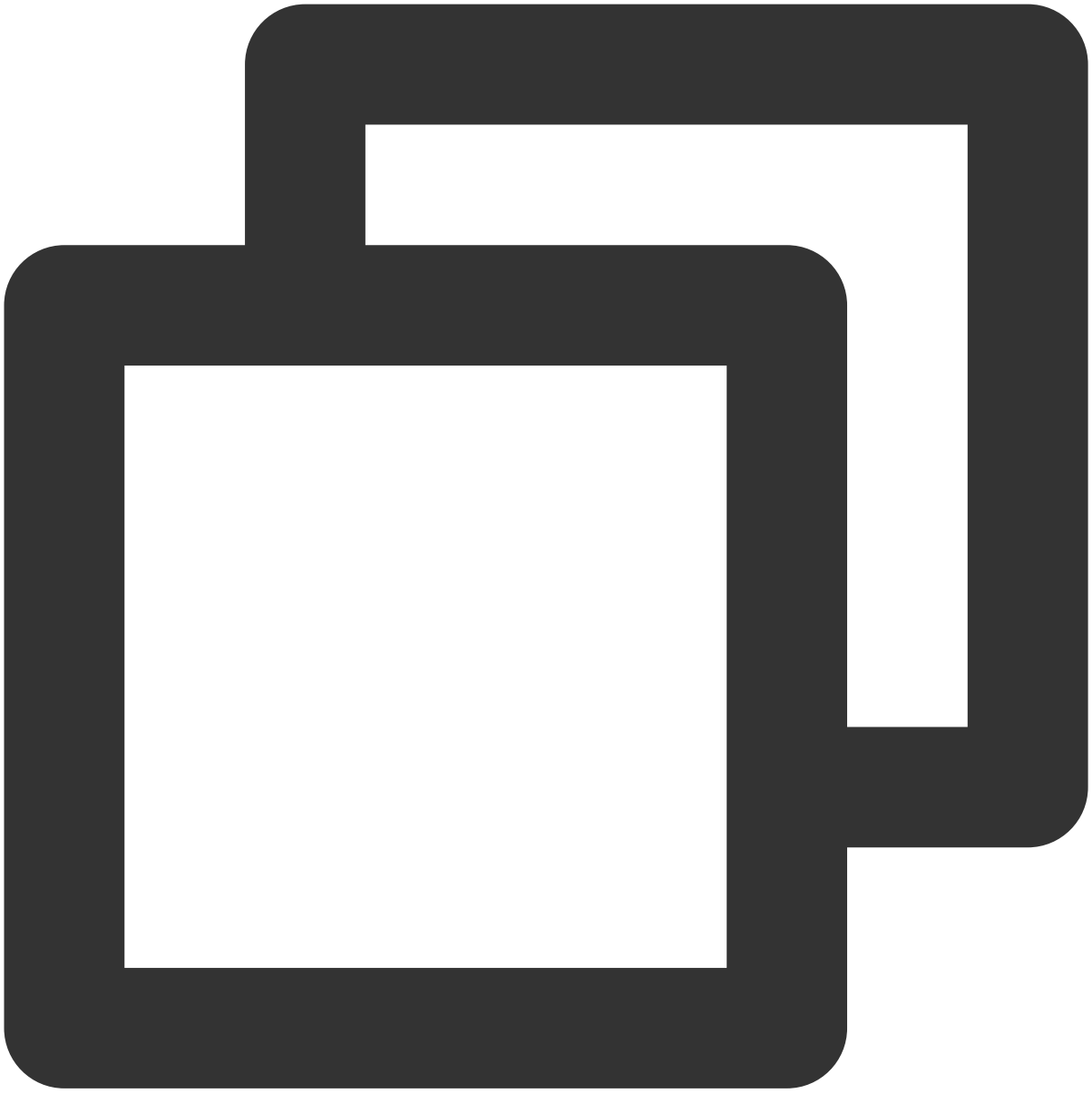
```
    LocalAgentHostPort: endPoint,
    },
    // Token configuration
    Tags:        []opentracing.Tag{ // Set the tag, where information such as token
    opentracing.Tag{Key: "token", Value: token}, // Set the token
    },
}
tracer, closer, err := cfg.NewTracer(jaegerConfig.Logger(jaeger.StdLogger)) // Get
```

3. Configure the interceptor.



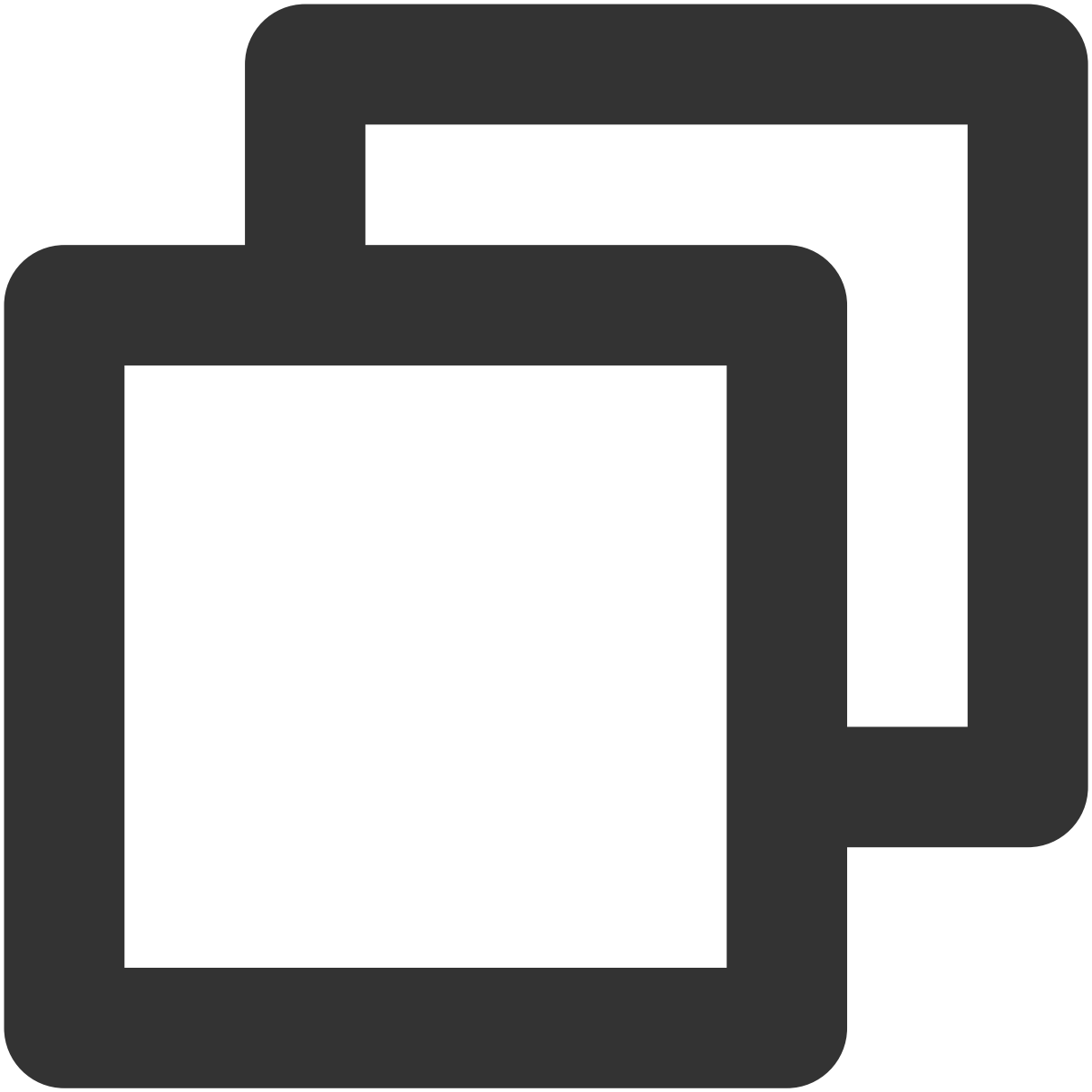
```
s := grpc.NewServer(grpc.UnaryInterceptor(otgrpc.OpenTracingServerInterceptor(trace
```

4. Start the server service.



```
// Register our service with the gRPC server
pb.RegisterHelloTraceServer(s, &server{})
if err := s.Serve(lis); err != nil {
   log.Fatalf("failed to serve: %v", err)
}
```

The complete code is as follows:



```
// Copyright © 2019-2020 Tencent Co., Ltd.

// This file is part of tencent project.
// Do not copy, cite, or distribute without the express
// permission from Cloud Monitor group.

package grpcdemo

import (
    "context"
    "fmt"
```

```
    "github.com/opentracing/opentracing-go"
    "github.com/uber/jaeger-client-go"
    jaegerConfig "github.com/uber/jaeger-client-go/config"
    "log"
    "net"

    "github.com/opentracing-contrib/go-grpc"
    "google.golang.org/grpc"
)

const (
    // Service name, which is the unique identifier of the service and the basis fo
    grpcServerName = "demo-grpc-server"
    serverPort     = ":9090"
)

// server is used to implement proto.HelloTraceServer.
type server struct {
    UnimplementedHelloTraceServer
}

// SayHello implements proto.HelloTraceServer
func (s *server) SayHello(ctx context.Context, in *TraceRequest) (*TraceResponse, e
    log.Printf("Received: %v", in.GetName())
    return &TraceResponse{Message: "Hello " + in.GetName()}, nil
}

// StartServer
func StartServer() {
    cfg := &jaegerConfig.Configuration{
        ServiceName: grpcServerName, // Call trace of the target service. Enter the
        Sampler: &jaegerConfig.SamplerConfig{ // Sampling policy configuration. See
            Type:  "const",
            Param: 1,
        },
        Reporter: &jaegerConfig.ReporterConfig{ // Configure how the client reports
            LogSpans:           true,
            LocalAgentHostPort: endPoint,
        },
        // Token configuration
        Tags: []opentracing.Tag{ // Set the tag, where information such as token ca
            opentracing.Tag{Key: "token", Value: token}, // Set the token
        },
    }
    tracer, closer, err := cfg.NewTracer(jaegerConfig.Logger(jaeger.StdLogger)) //
    defer closer.Close()
    if err != nil {
```

```
        panic(fmt.Sprintf("ERROR: fail init Jaeger: %v\\n", err))
    }
    lis, err := net.Listen("tcp", serverPort)
    if err != nil {
        log.Fatalf("failed to listen: %v", err)
    }
    s := grpc.NewServer(grpc.UnaryInterceptor(otgrpc.OpenTracingServerInterceptor(t

    // Register our service with the gRPC server
    RegisterHelloTraceServer(s, &server{})
    if err := s.Serve(lis); err != nil {
        log.Fatalf("failed to serve: %v", err)
    }
}
```

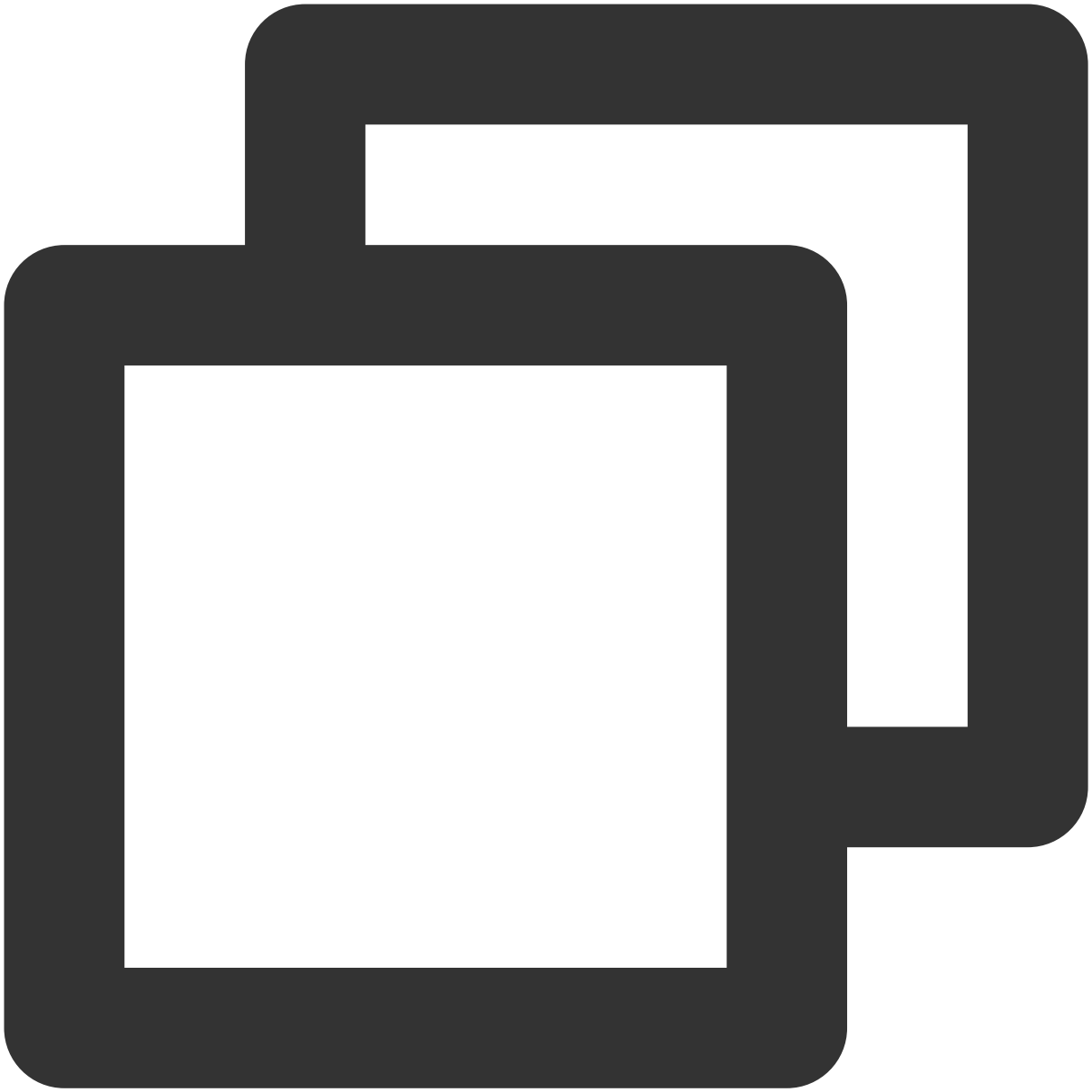
Client

1. Due to the need to simulate HTTP requests on the client, import the `opentracing-contrib/go-stdlib/nethttp` dependency.

Dependency path: `github.com/opentracing-contrib/go-stdlib/nethttp`

Version requirement: ≥ `v1.0.0`

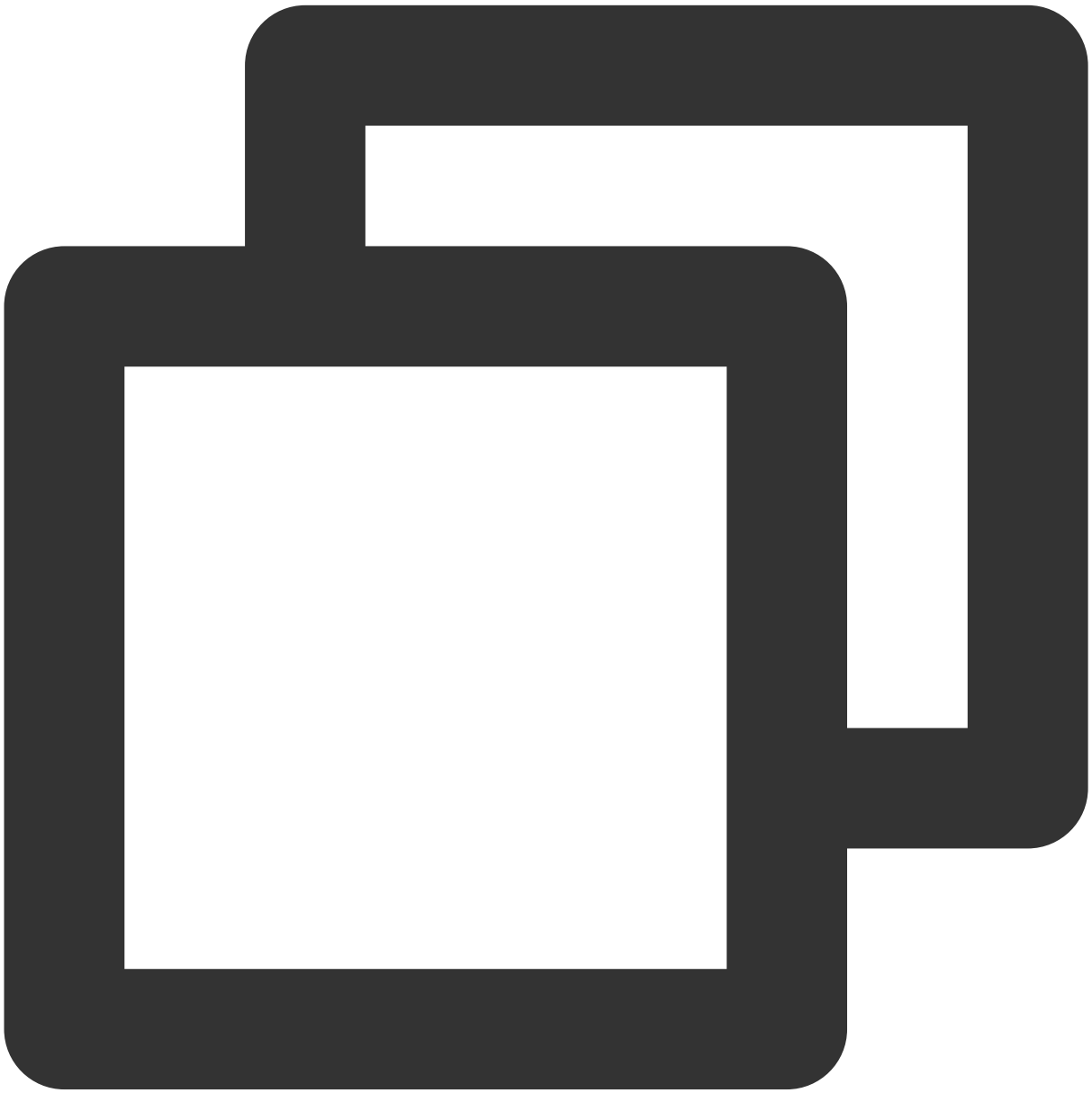
2. Configure Jaeger and create a trace object.



```
cfg := &jaegerConfig.Configuration{
    ServiceName: grpcClientName, // Call trace of the target service. Enter the ser
    Sampler: &jaegerConfig.SamplerConfig{ // Sampling policy configuration. See sec
    Type:  "const",
    Param: 1,
    },
    Reporter: &jaegerConfig.ReporterConfig{ // Configure how the client reports tra
    LogSpans:          true,
    LocalAgentHostPort: endPoint,
    },
    // Token configuration
```

```
    Tags:        []opentracing.Tag{ // Set the tag, where information such as token
    opentracing.Tag{Key: "token", Value: token}, // Set the token
    },
}
tracer, closer, err := cfg.NewTracer(jaegerConfig.Logger(jaeger.StdLogger)) // Get
```

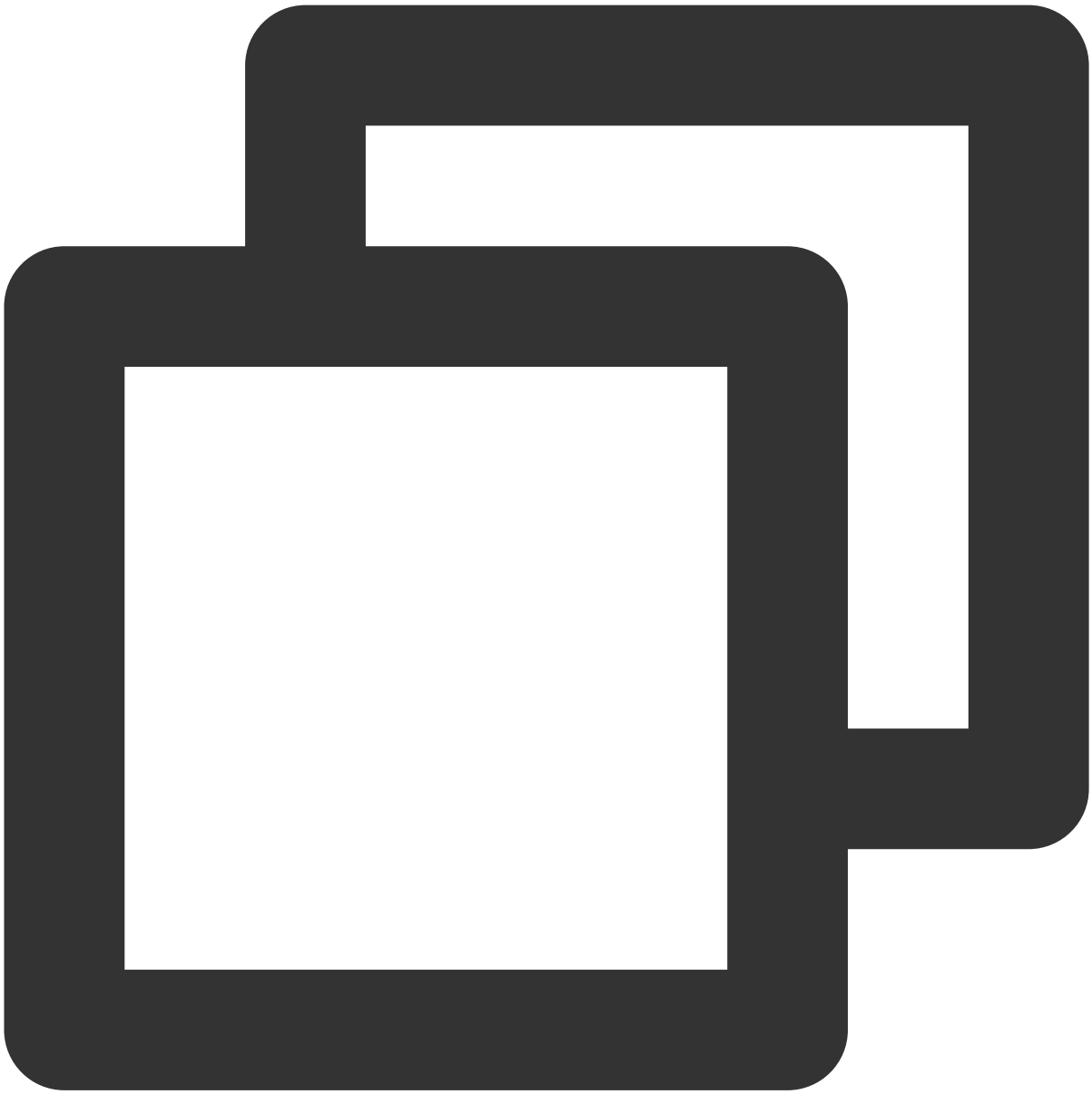
3. Establish a connection to configure the interceptor.



```
// Establish a connection to the server to configure the interceptor
conn, err := grpc.Dial(serverAddress, grpc.WithInsecure(), grpc.WithBlock(),
       grpc.WithUnaryInterceptor(otgrpc.OpenTracingClientInterceptor(tracer)))
```

4. Make a gRPC call to check whether the access is successful.

The complete code is as follows:



```
// Copyright © 2019-2020 Tencent Co., Ltd.

// This file is part of tencent project.
// Do not copy, cite, or distribute without the express
// permission from Cloud Monitor group.

package grpcdemo
```

```
import (
    "context"
    "fmt"
    "github.com/opentracing-contrib/go-grpc"
    "github.com/opentracing/opentracing-go"
    "github.com/uber/jaeger-client-go"
    jaegerConfig "github.com/uber/jaeger-client-go/config"
    "google.golang.org/grpc"
    "log"
    "time"
)

const (
    // Service name, which is the unique identifier of the service and the basis fo
    grpcClientName = "demo-grpc-client"
    defaultName    = "TAW Tracing"
    serverAddress  = "localhost:9090"
    endPoint       = "xxxxx:6831" // Local agent address
    token          = "abc"
)

// StartClient
func StartClient() {
    cfg := &jaegerConfig.Configuration{
        ServiceName: grpcClientName, // Call trace of the target service. Enter the
        Sampler: &jaegerConfig.SamplerConfig{ // Sampling policy configuration. See
            Type:  "const",
            Param: 1,
        },
        Reporter: &jaegerConfig.ReporterConfig{ // Configure how the client reports
            LogSpans:           true,
            LocalAgentHostPort: endPoint,
        },
        // Token configuration
        Tags: []opentracing.Tag{ // Set the tag, where information such as token ca
            opentracing.Tag{Key: "token", Value: token}, // Set the token
        },
    }
    tracer, closer, err := cfg.NewTracer(jaegerConfig.Logger(jaeger.StdLogger)) //
    defer closer.Close()
    if err != nil {
        panic(fmt.Sprintf("ERROR: fail init Jaeger: %v\\n", err))
    }
    // Establish a connection to the server to configure the interceptor
    conn, err := grpc.Dial(serverAddress, grpc.WithInsecure(), grpc.WithBlock(),
        grpc.WithUnaryInterceptor(otgrpc.OpenTracingClientInterceptor(tracer)))
    if err != nil {
```

```
        log.Fatalf("did not connect: %v", err)
    }
    defer conn.Close()

    //
    c := NewHelloTraceClient(conn)
    ctx, cancel := context.WithTimeout(context.Background(), time.Second)
    defer cancel()
    // Make an RPC call
    r, err := c.SayHello(ctx, &TraceRequest{Name: defaultName})
    if err != nil {
        log.Fatalf("could not greet: %v", err)
    }
    log.Printf("RPC Client receive: %s", r.GetMessage())
}
```

Accessing Java Application
Automatic Connecting Java Application for the TKE Environment (Recommended)

Last updated：2024-06-19 16:31:30

For Java Applications deployed on TKE, APM offers an automatic connection scheme, enabling agent automatic injection after the application is deployed to TKE, facilitating quick connecting.
Java applications that are automatically connected in the TKE environment will be automatically injected with the Tencent Cloud OpenTelemetry Java Agent Enhanced Edition (TencentCloud-OTel Java Agent). The Tencent Cloud OpenTelemetry Java Agent Enhanced Edition is based on the secondary development of OpenTelemetry-java-instrumentation from the open-source community, adhering to the Apache License 2.0 protocol. The OpenTelemetry License is cited within the agent packet. Building upon the open-source agent, the Tencent Cloud OpenTelemetry Java Agent Enhanced Edition has improved significantly in Event Tracking density, advanced diagnosis, performance protection, and enterprise-level capabilities.

Prerequisites

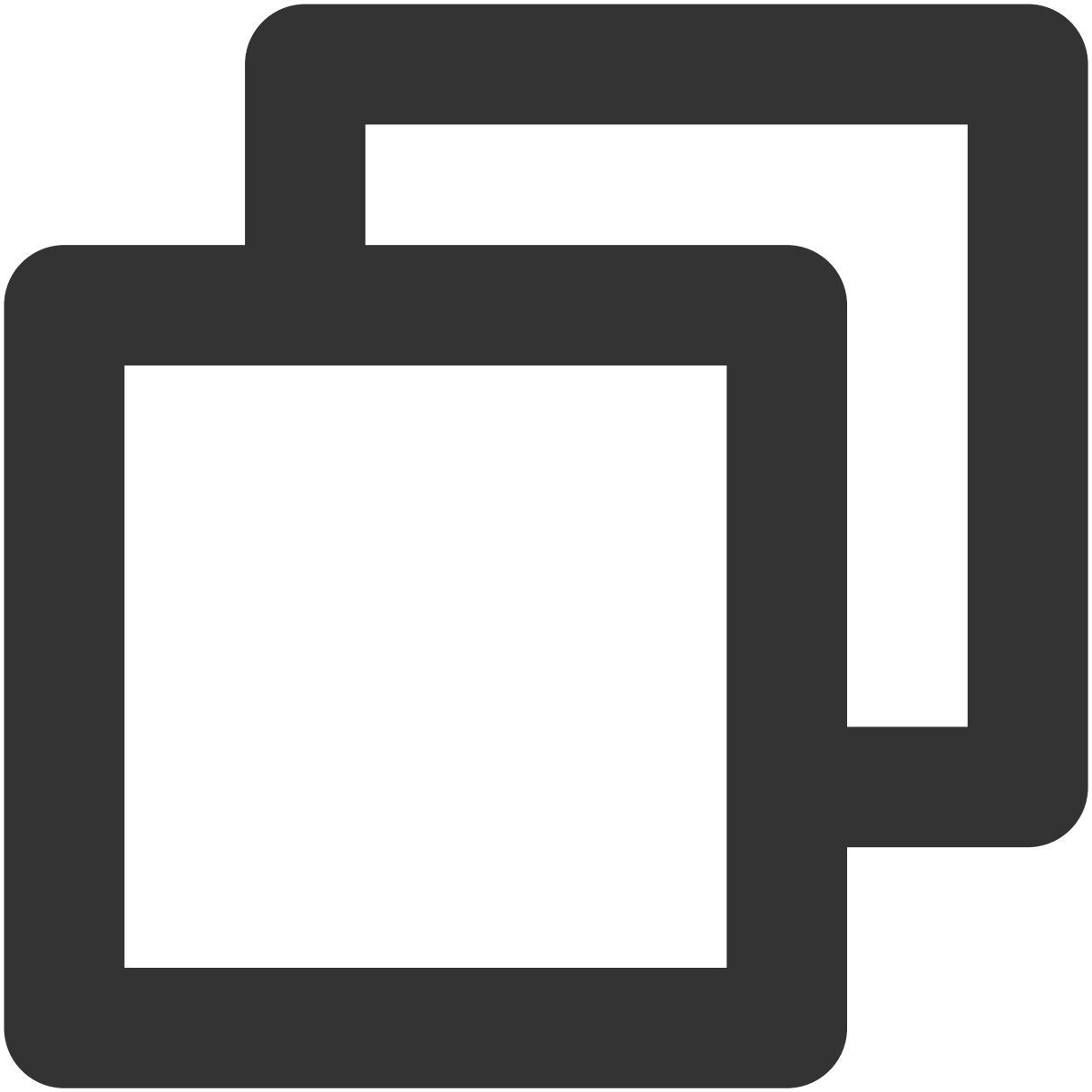
See Supported Java Versions and Frameworks by the OTel Java Agent Enhanced Edition to ensure that the Java version and application server are within the supported range by the agent. For the dependency libraries and frameworks supported by automatic tracing, data reporting can be completed upon successful connection without modifying the code. In addition, the Tencent Cloud OpenTelemetry Java Agent Enhanced Edition adheres to the OpenTelemetry protocol standard. if automatic Event Tracking does not meet your scenario, or you need to add business layer Event Tracking, use OpenTelemetry API for Custom Metrics Definition.

Step 1: Install Operator.

Install Operator in the TKE cluster, it's recommended to install Operator with one click on the APM console, for details see installing tencent-opentelemetry-operator.

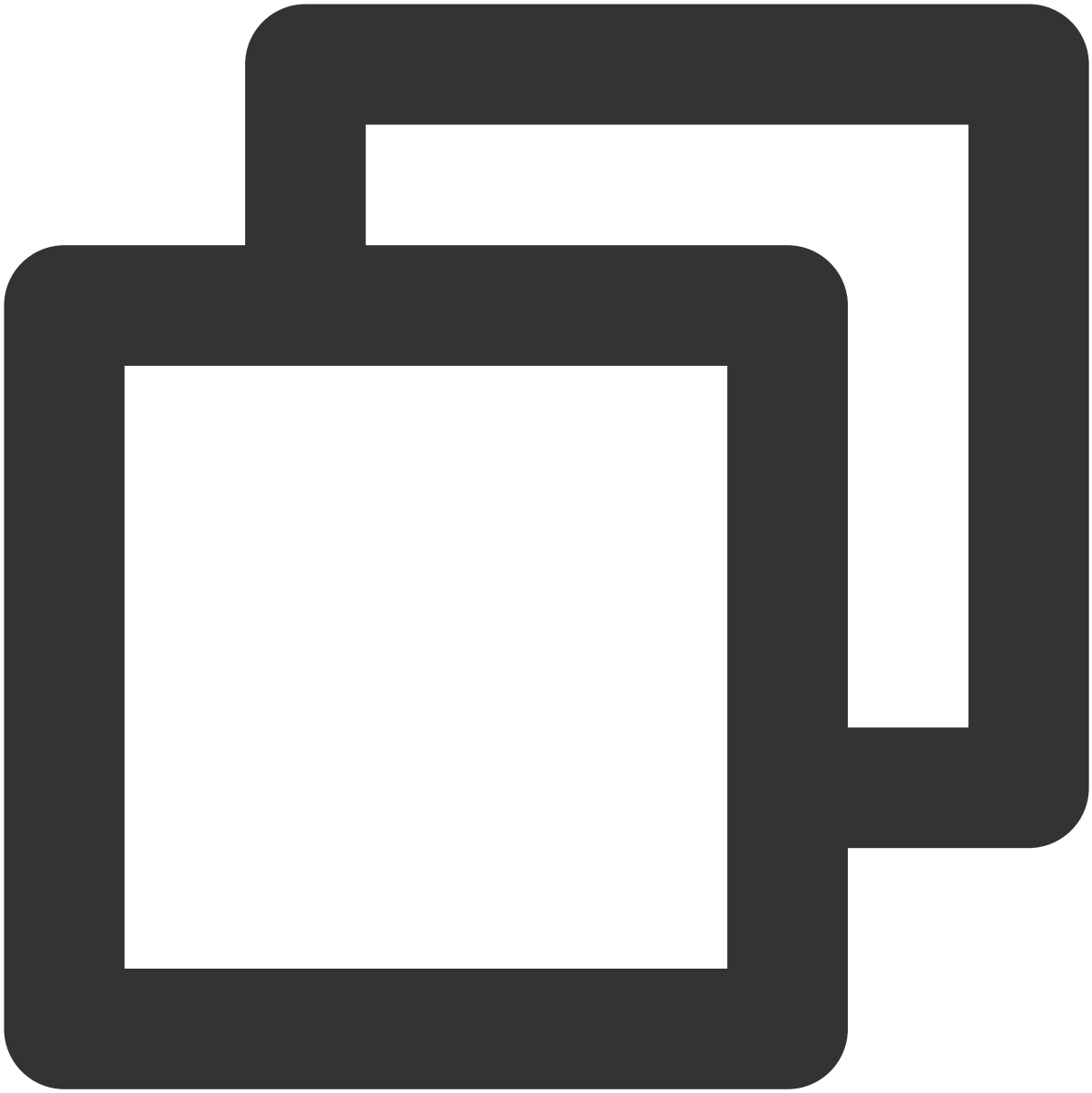
Step 2: Add annotation to workload.

1. Log in to TKE console.
2. Click **Cluster** to enter the corresponding TKE cluster.
3. In **Workload**, you can find the application that needs to connect APM, click **More**, then click **Edit YAML**.
4. Apply the following content in the Pod annotation, then click **Complete** to finish the connection.



```
cloud.tencent.com/inject-java: "true"
cloud.tencent.com/otel-service-name: my-app  # Application name. Processes of conne
# The application name can be up to 63 characters and can only contain lowercase le
```

Note that this content needs to be added to `spec.template.metadata.annotations`, affecting the Pod's annotation, not the workload's annotation. You can see the following code snippet:



```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    k8s-app: my-app
  name: my-app
  namespace: default
spec:
  selector:
    matchLabels:
      k8s-app: my-app
```

```
  template:
    metadata:
      labels:
        k8s-app: my-app
      annotations:
        cloud.tencent.com/inject-java: "true"  # Add it here.
        cloud.tencent.com/otel-service-name: my-app
    spec:
      containers:
        image: my-app:0.1
        name: my-app
```

Connection Verification

After annotations are added to the workload, based on different publish policies, you can trigger a restart of the application pod. The newly launched pod will automatically inject the agent and connect to the APM server to report monitoring data, with the reported business system being the default business system of the Operator. In normal traffic cases, the connected application will displayed in APM > Application monitoring > Application list and the connected application instances will be displayed in APM > Application Monitoring > **App details > Instance monitoring**. Since there is a certain latency in the processing of observable data, if the application or instance does not appear in the console after connecting, wait for about 30 seconds.

Custom Event Tracking

When automatic instrumentation does not meet your scenario, or you need to add business layer instrumentation, see Custom Event Tracking and use the OpenTelemetry API to add custom instrumentation.

More Connection Configuration Items (Optional)

At the Workload level, you can add more annotations to adjust the connect behaviors:

Configuration Item	Description
cloud.tencent.com/apm-token	Specify the Token for the APM business system. If this configuration item is not added, the Operator's configuration is used (corresponding to the Operator's `env.APM_TOKEN` field).
cloud.tencent.com/java-instr-version	Specify the Java agent version. If this configuration item is not added, the configuration of the Operator is used (corresponding to the Operator's `env.JAVA_INSTR_VERSION` field). The value can be `latest` (default) or a specific version number. For a list of specific version numbers, see Agent Version Information. It is not recommended to fill in this field unless necessary.

Connecting via Tencent Cloud OpenTelemetry Java Agent Enhanced Edition (Recommended)

Last updated：2024-06-19 16:31:30

Tencent Cloud OpenTelemetry Java Agent Enhanced Edition is based on the OpenTelemetry-java-instrumentation from the open-source community and developed further under the Apache License 2.0 protocol. It includes a reference to the OpenTelemetry License within the agent packet. Building on the open source agent, the Tencent Cloud OpenTelemetry Java Agent Enhanced Edition has significantly improved in Event Tracking density, advanced diagnosis, performance protection, and enterprise-level capabilities.

Note:

OpenTelemetry is a collection of tools, APIs, and SDKs for monitoring, generating, collecting, and exporting telemetry data (metrics, logs, and traces) to help users analyze the performance and behavior of the software. For more information about OpenTelemetry, see the OpenTelemetry official website.

The OpenTelemetry community is active, with rapid technological changes, and widely compatible with mainstream programming languages, components, and frameworks, making its link-tracing capability highly popular for cloud-native microservices and container architectures.

This document will guide you on how to connect Java applications with the Tencent Cloud OpenTelemetry Agent Enhanced Edition using related operations.

Prerequisites

See Supported Java Versions and Frameworks by OTel Java Agent Enhanced Edition to ensure that your Java version and application server are within the supported range of the agent. For dependency libraries and frameworks supported by automatic Event Tracking, data reporting can be completed upon successful connection without modifying the code. In addition, the Tencent Cloud OpenTelemetry Java Agent Enhanced Edition complies with the OpenTelemetry protocol standards. If automatic Event Tracking does not meet your scenario, or you need to add business layer Event Tracking, see Custom Event Tracking and use the OpenTelemetry API for custom Event Tracking.

Step 1. Get the connect point and Token.

1. Log in to the TCOP console.

2. In the left menu column, select **Application Performance Management > Application monitoring**, and click **Application list > Access application**.
3. In the **Data Access** drawer that appears on the right, click **Java** language.
4. On the **Access Java application** page, select the **region** and **business system** you want to connect.
5. Select **Access protocol type** as **OpenTelemetry**.
6. **Reporting method** Choose your desired reporting method, and obtain your **Access Point** and **Token**.

Note:

Private network reporting: Using this reporting method, your service needs to run in the Tencent Cloud VPC. Through VPC connecting directly, you can avoid the security risks of public network communication and save on reporting traffic overhead.

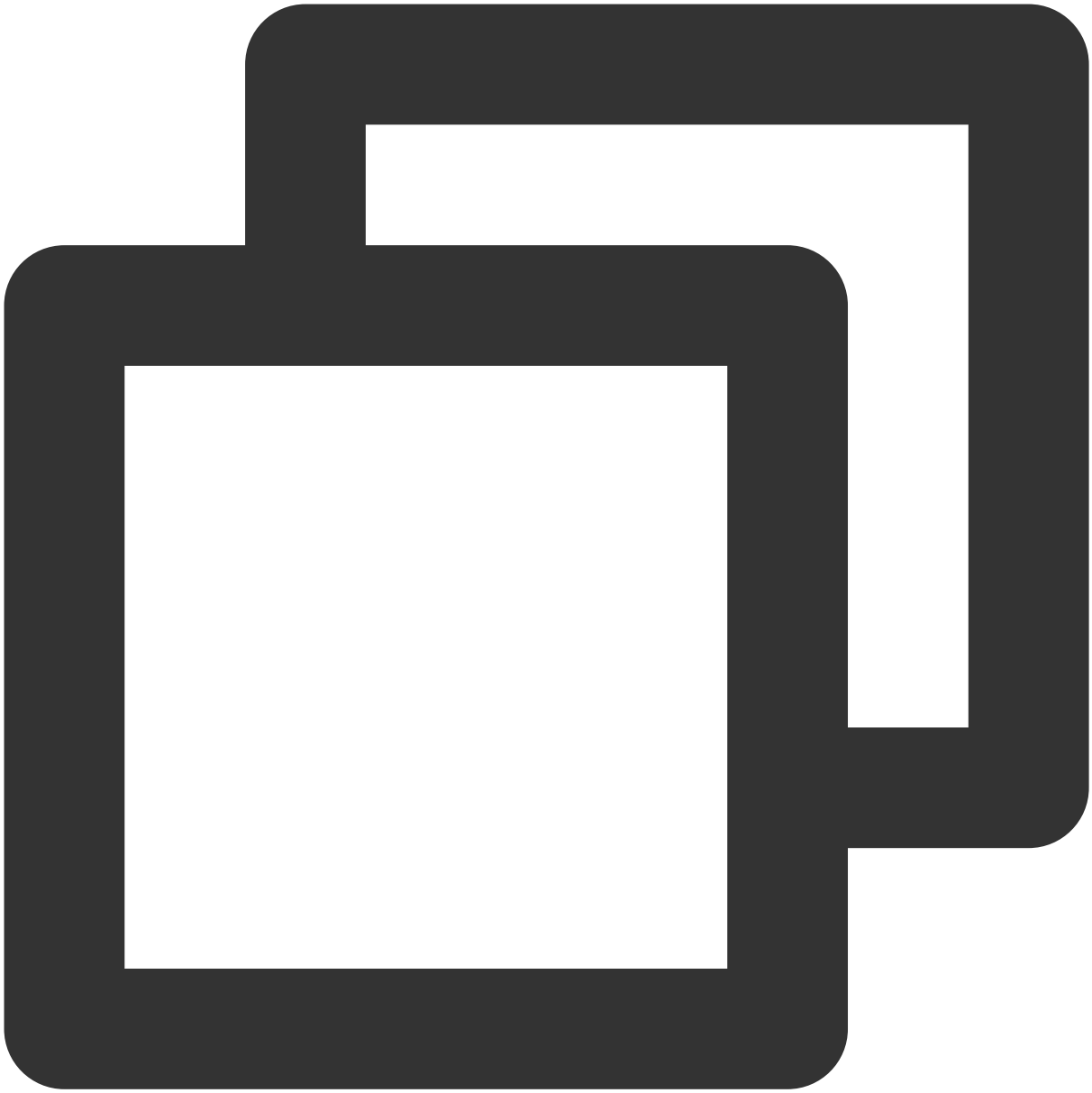
Public network reporting: If your service is deployed locally or in non-Tencent Cloud VPC, you can report data in this method. However, it involves security risks in public network communication and incurs reporting traffic fees.

Step 2: Download the agent.

Go to Agent Version Information to download the agent, it is recommended to download the latest version, named `opentelemetry-javaagent.jar` .

Step 3: Modify the reporting parameters.

Connecting Java applications requires the following 3 JVM startup parameters:



```
-javaagent:<javaagent>
-Dotel.resource.attributes=service.name=<serviceName>,token=<token>
-Dotel.exporter.otlp.endpoint=<endpoint>
```

When you execute the Java command, ensure these 3 JVM startup parameters are placed before the `-jar` . For applications that cannot directly specify JVM startup parameters, the system parameter `-Dotel.resource.attributes` can be replaced with the environment variable `OTEL_RESOURCE_ATTRIBUTES` , and the system parameter `-Dotel.exporter.otlp.endpoint` can be

replaced with the environment variable `OTEL_EXPORTER_OTLP_ENDPOINT`. The corresponding fields are explained as follows:

`<javaagent>`: the local file path of the agent.

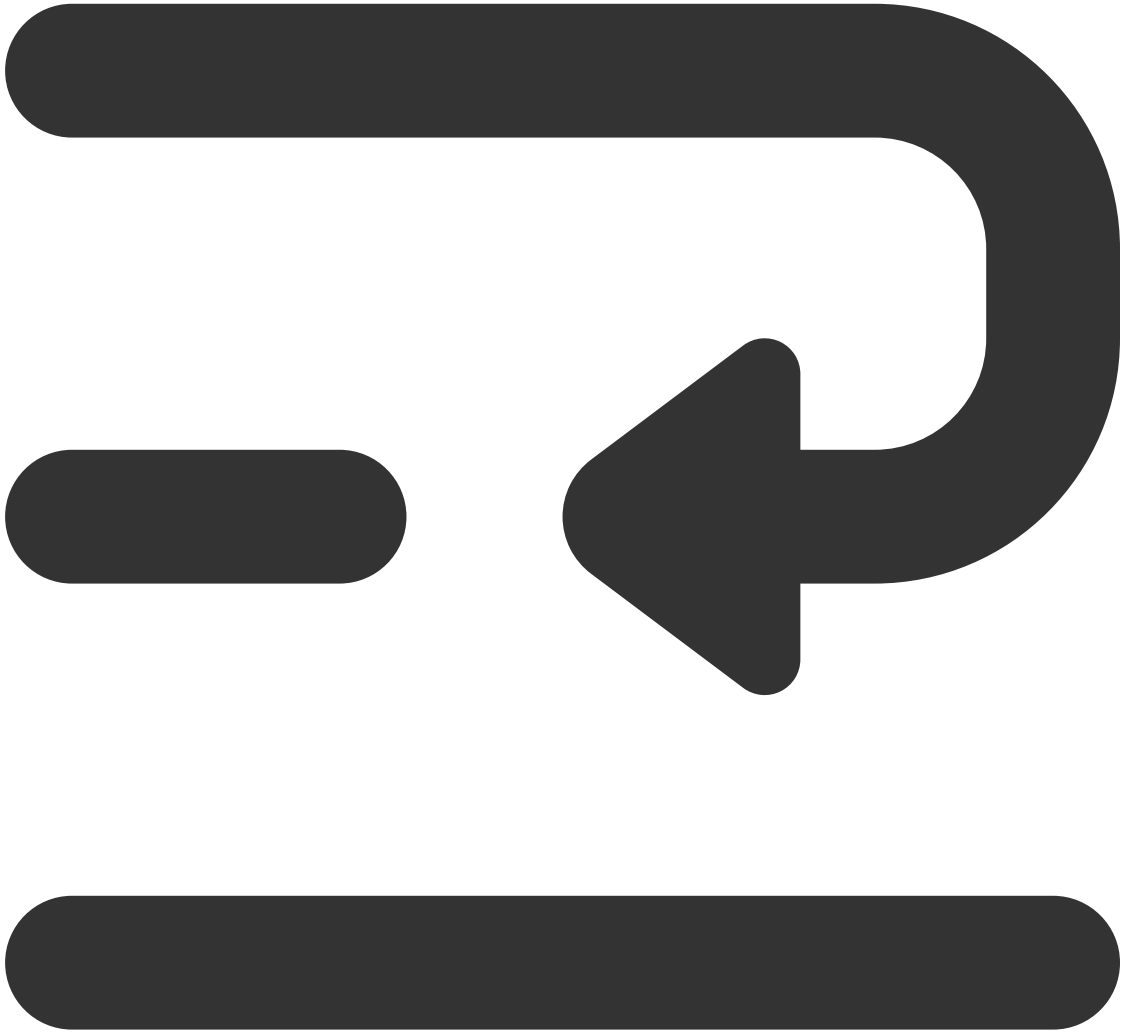
`<serviceName>`: Application name. Processes of connecting using the same application name are displayed as multiple instances under the same application in APM. For Spring Cloud or Dubbo applications, the application name usually matches the service name. It can be up to 63 characters and can only contain lowercase letters, digits, and the separator (-), and it must start with a lowercase letter and end with a digit or lowercase letter.

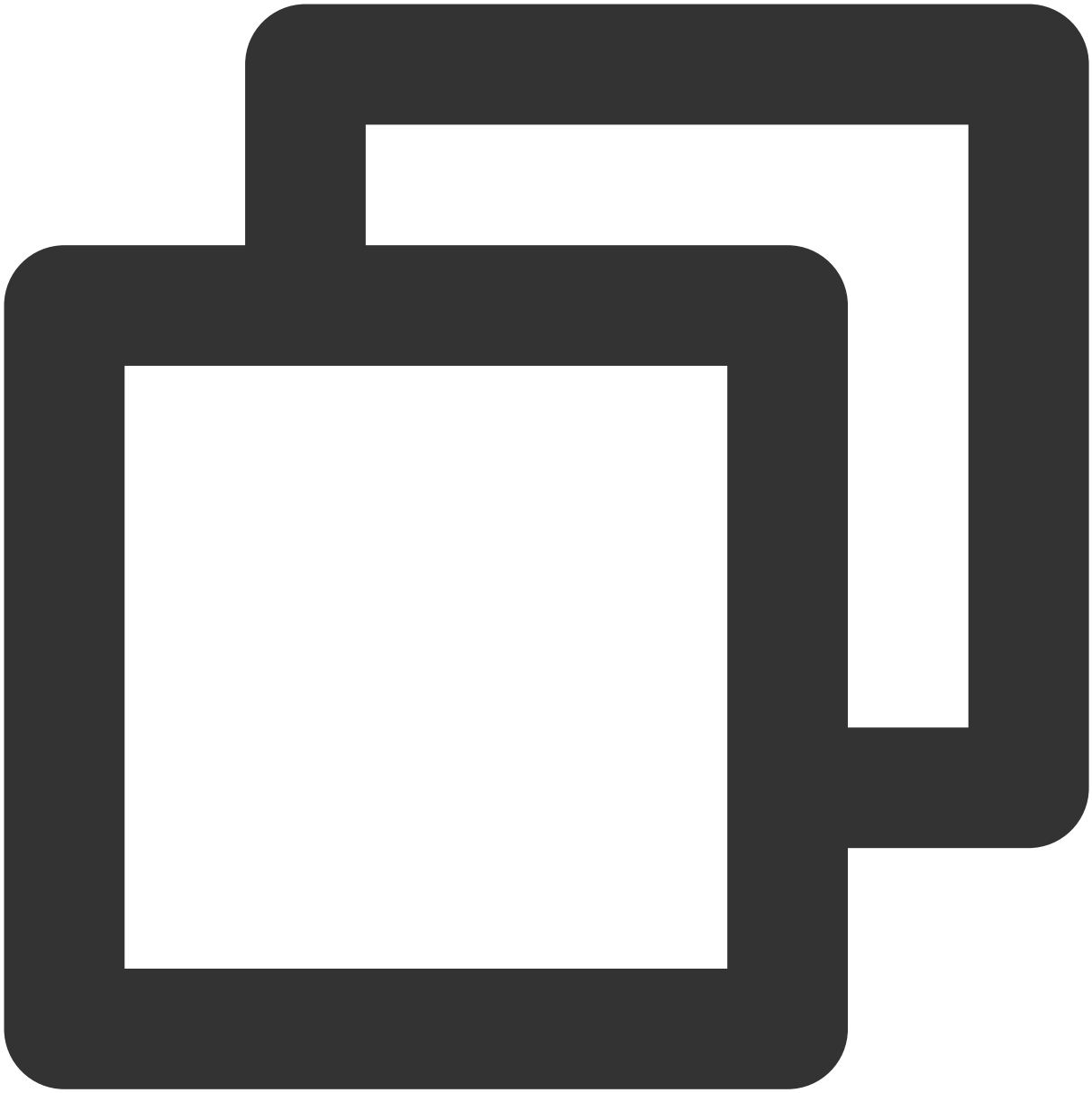
`<token>`: the business system Token obtained in step 1.

`<endpoint>`: the connect point obtained in step 1.

The following content uses the agent path `/path/to/opentelemetry-javaagent.jar`, the application name `myService`, the business system Token myToken, and the connect point `http://pl-demo.ap-guangzhou.apm.tencentcs.com:4317` as examples to introduce the complete start-up scripts for different environments:

JAR File or Spring Boot

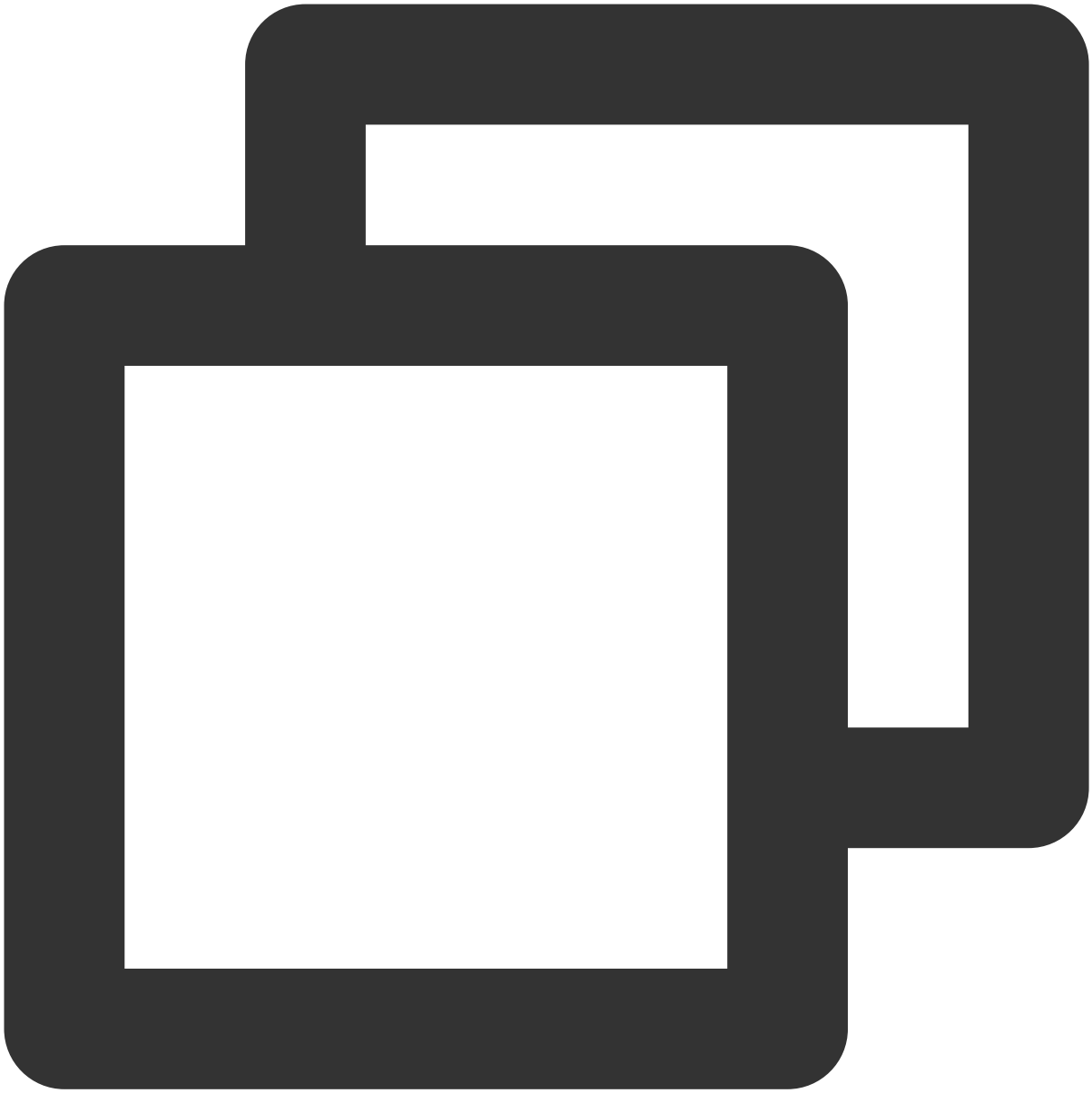




```
java -javaagent:/path/to/opentelemetry-javaagent.jar \\
-Dotel.resource.attributes=service.name=myService,token=myToken\\
-Dotel.exporter.otlp.endpoint=http://pl-demo.ap-guangzhou.apm.tencentcs.com:4317 \\
-jar SpringCloudApplication.jar
```

Linux Tomcat 7/Tomcat 8

Add the following content to the `{TOMCAT_HOME}/bin/setenv.sh` configuration file:

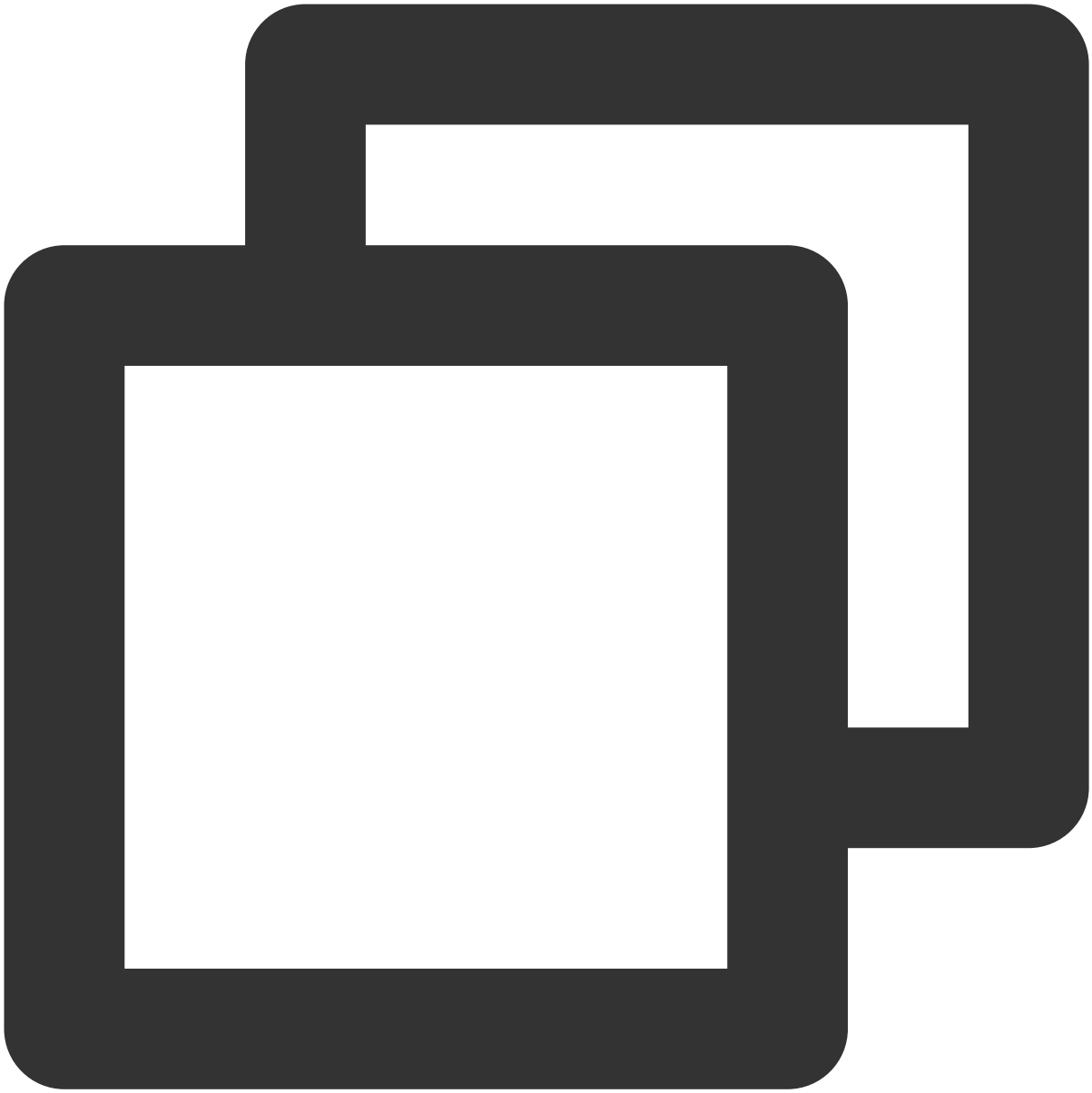


```
CATALINA_OPTS="$CATALINA_OPTS -javaagent:/path/to/opentelemetry-javaagent.jar"
export OTEL_RESOURCE_ATTRIBUTES=service.name=myService,token=myToken
export OTEL_EXPORTER_OTLP_ENDPOINT=http://pl-demo.ap-guangzhou.apm.tencentcs.com:43
```

If your Tomcat does not have a setenv.sh configuration file, see the Tomcat official documentation to initialize a setenv.sh configuration file, or use another method to add Java start-up parameters and environment variables.

Jetty

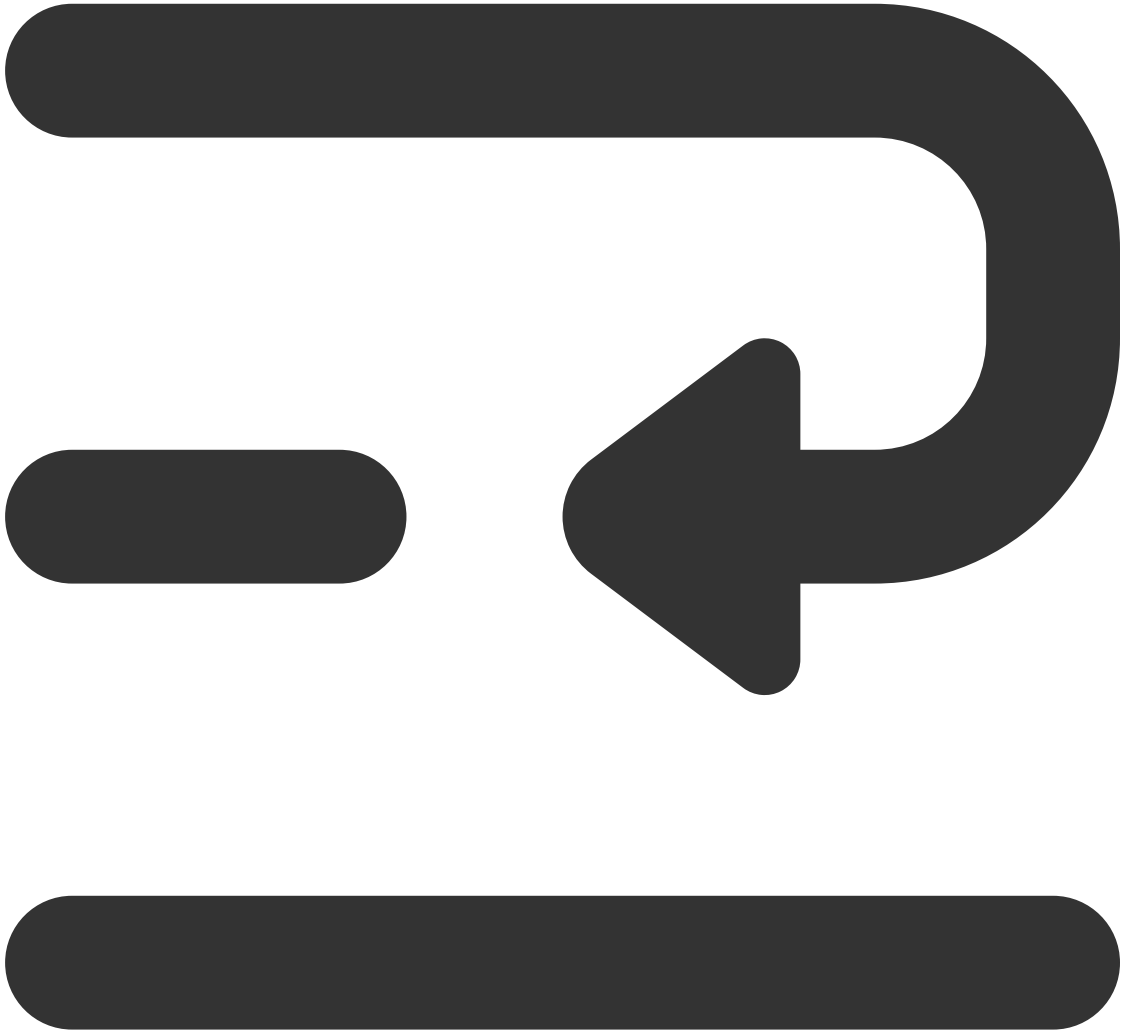
Add the following content in the `<jetty_home\\>/bin/jetty.sh` start-up script:

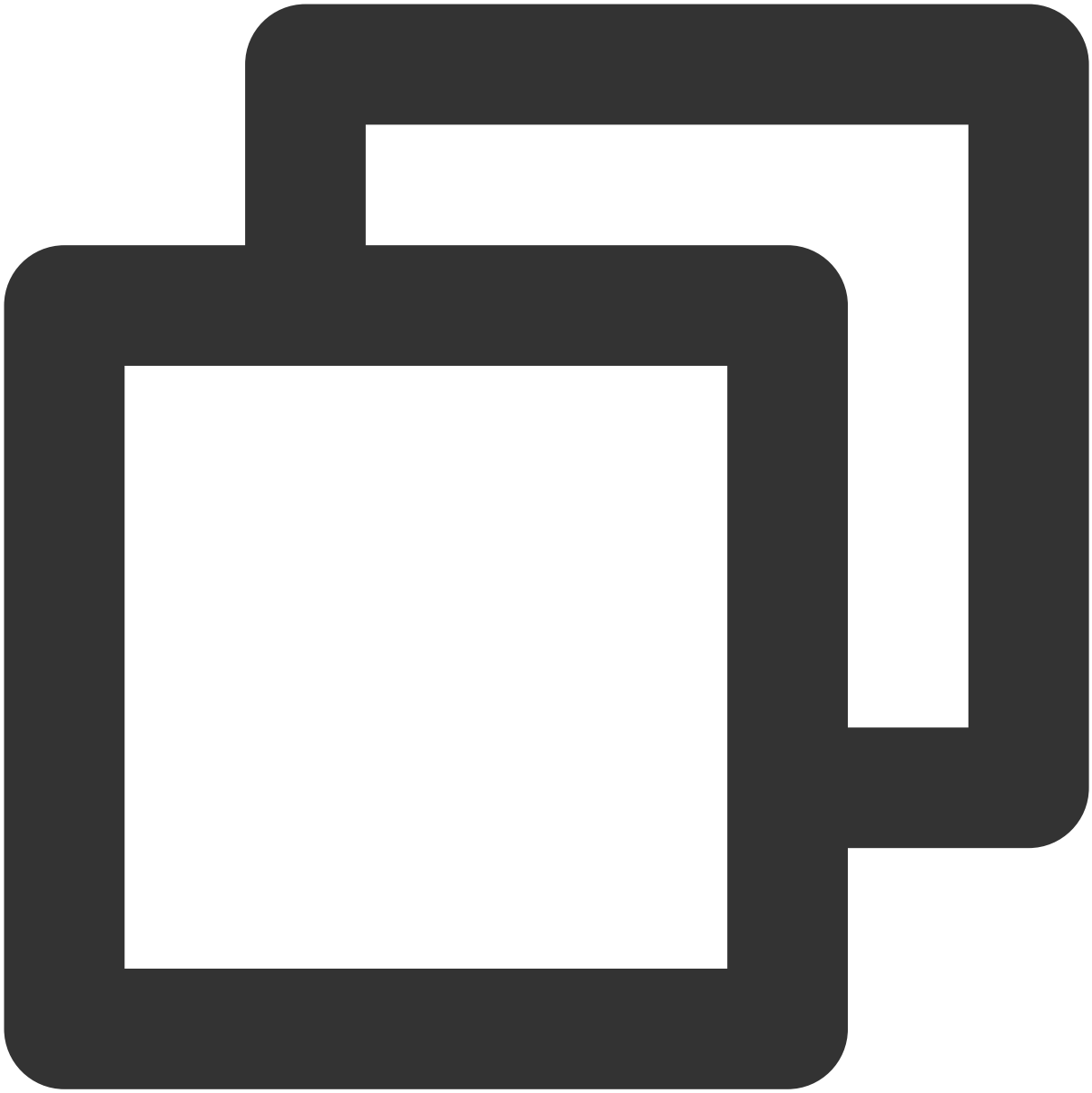


```
JAVA_OPTIONS="$JAVA_OPTIONS -javaagent:/path/to/opentelemetry-javaagent.jar"
export OTEL_RESOURCE_ATTRIBUTES=service.name=myService,token=myToken
export OTEL_EXPORTER_OTLP_ENDPOINT=http://pl-demo.ap-guangzhou.apm.tencentcs.com:43
```

IDEA

When you debug Java applications locally in IDEA, you can configure VM options in the Run Configuration, with parameters as follows:





```
-javaagent:"/path/to/opentelemetry-javaagent.jar"
-Dotel.resource.attributes=service.name=myService,token=myToken
-Dotel.exporter.otlp.endpoint=http://pl-demo.ap-guangzhou.apm.tencentcs.com:4317
```

In this case, ensure network connectivity between the local environment and the connect point. Usually, the public network connect point address can be used for reporting.

Other application servers

See the corresponding configuration standards to mount the agent, and add Java start-up parameters or environment variables.

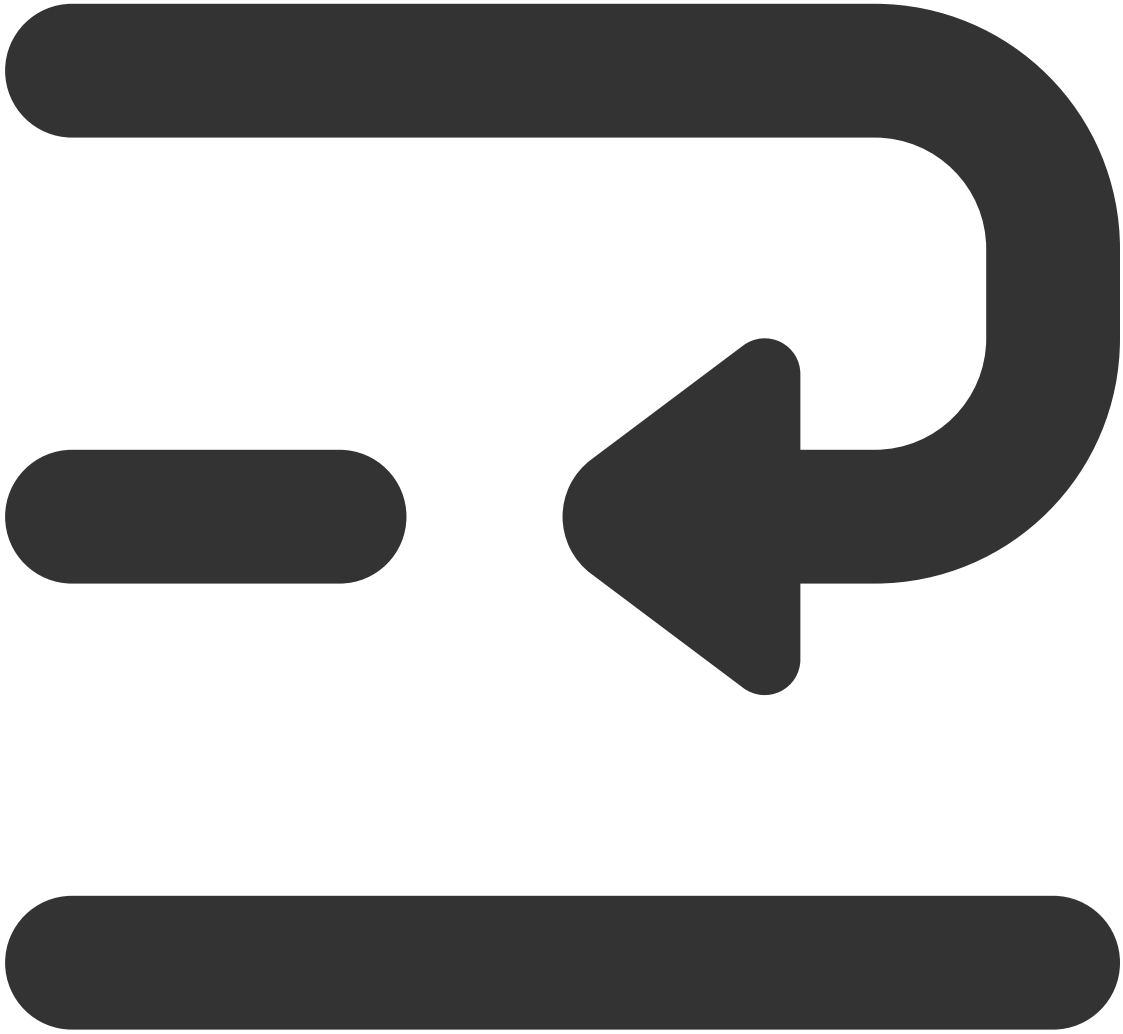
Connection Verification

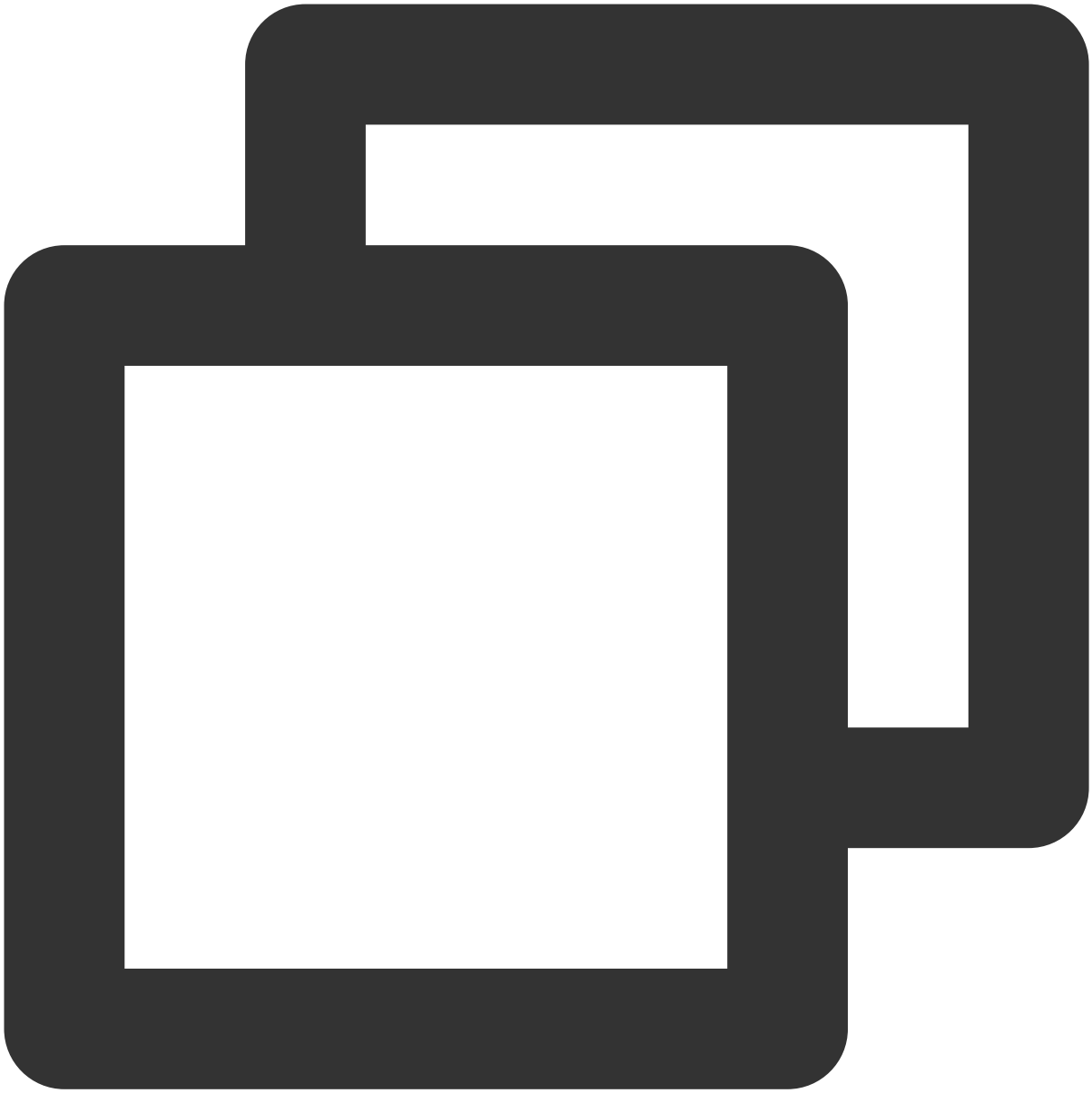
After the 3 connect steps are completed, start the Java application. The application will mount the agent and connect to the APM server to report monitoring data. In normal traffic cases, the connected applications will be displayed in the console APM > Application monitoring > Application list , and the connected application instances will be displayed in the console **Application details > Instance monitoring**. Due to the delay in processing observable data, if the application or instance is not found in the console after connecting, wait for about 30 seconds.

Custom Event Tracking (Optional)

When automatic Event Tracking does not meet your scenario, or you need to add business layer Event Tracking, you can see the following content and use the OpenTelemetry API to add custom Event Tracking. This document only showcases the most basic way of custom Event Tracking, and the OpenTelemetry community provides more flexible ways of custom Event Tracking. For specific usage, you can see the OpenTelemetry official documentation.

Introducing OpenTelemetry API Dependencies



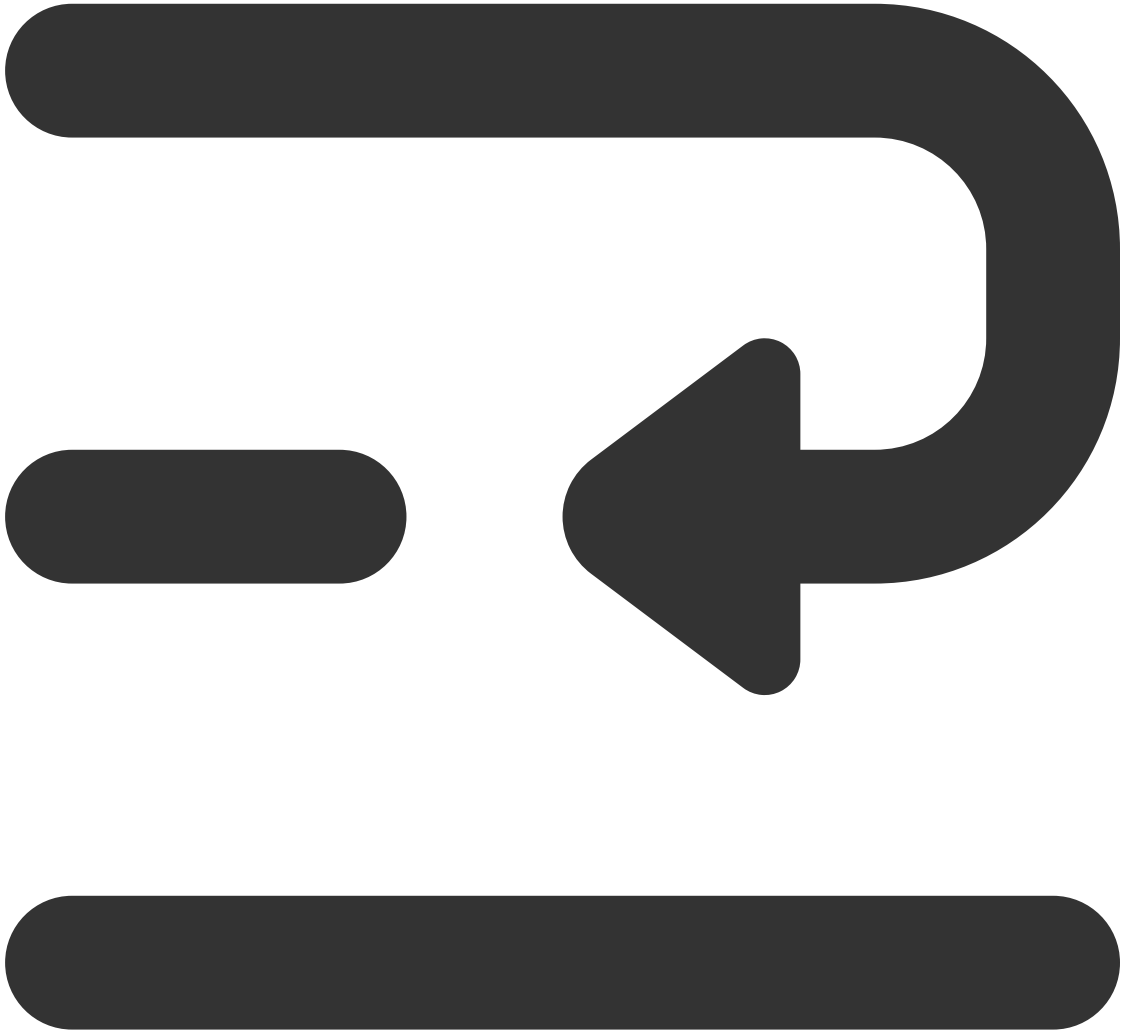


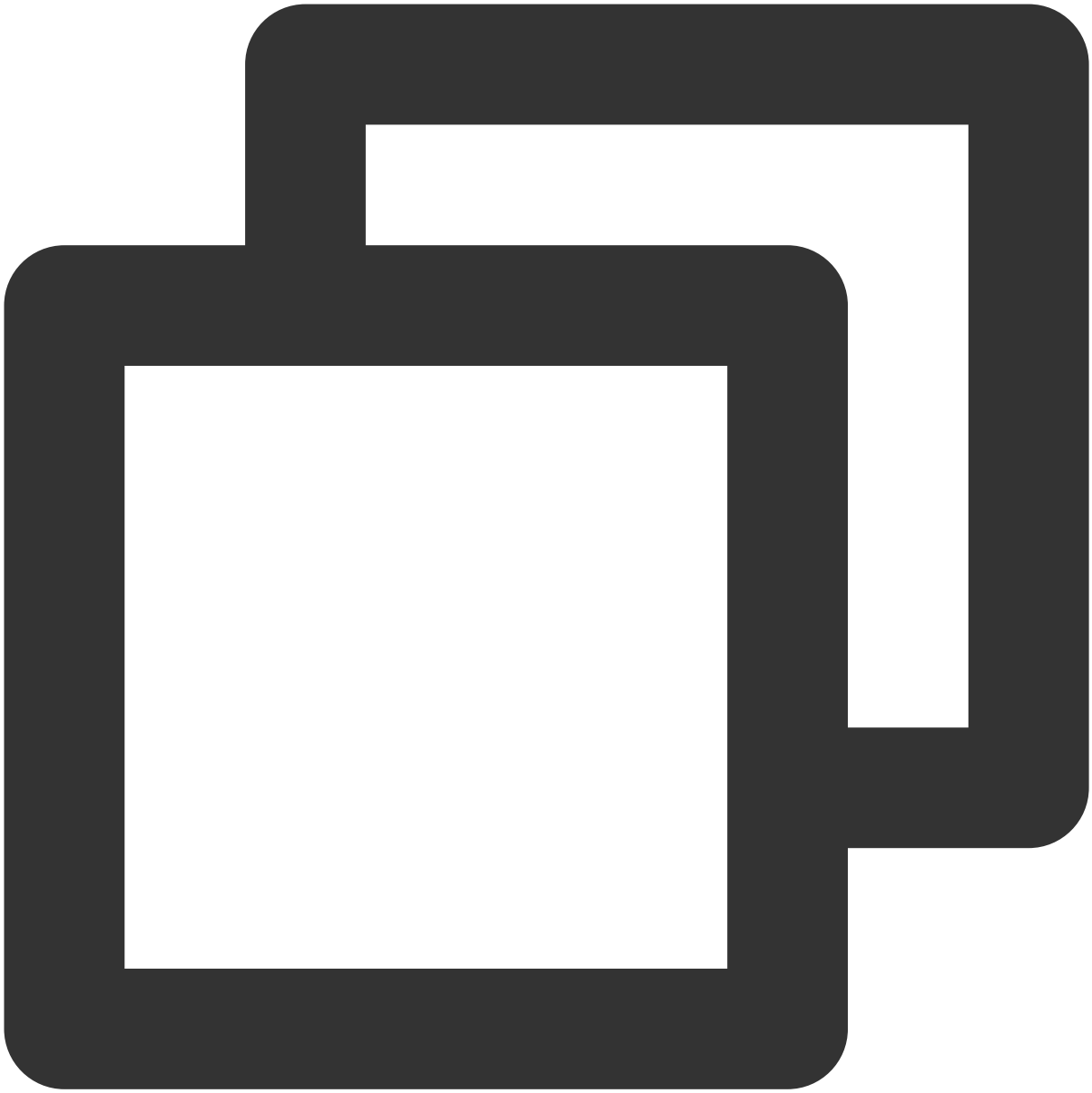
```
<dependencies>
    <!-- Other dependencies -->
    <dependency>
        <groupId>io.opentelemetry</groupId>
        <artifactId>opentelemetry-api</artifactId>
    </dependency>
</dependencies>
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>io.opentelemetry</groupId>
```

```
            <artifactId>opentelemetry-bom</artifactId>
            <version>1.9.0</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

Obtaining Tracer

In the code where Event Tracking needs to be implemented, the Tracer object can be obtained with the following code:

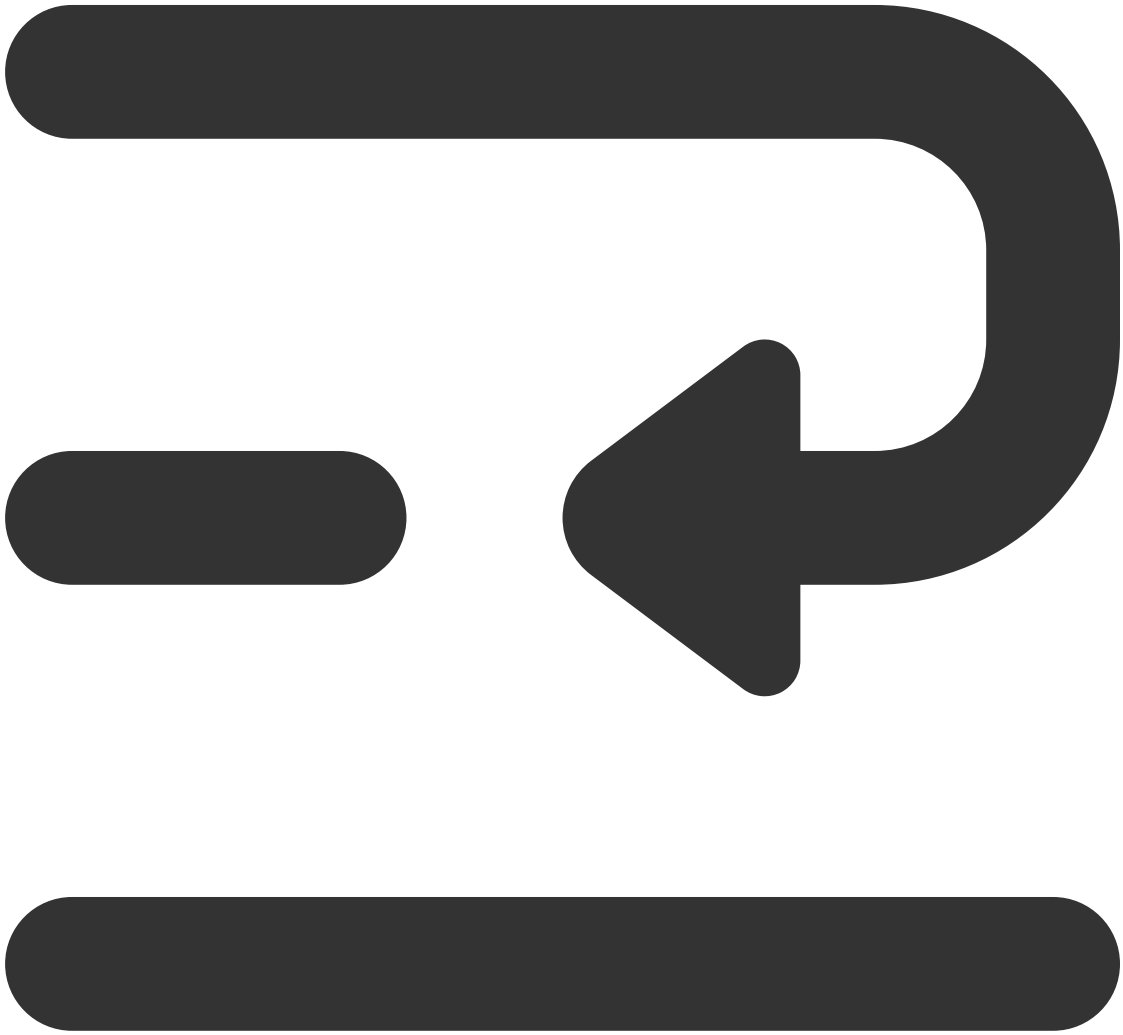


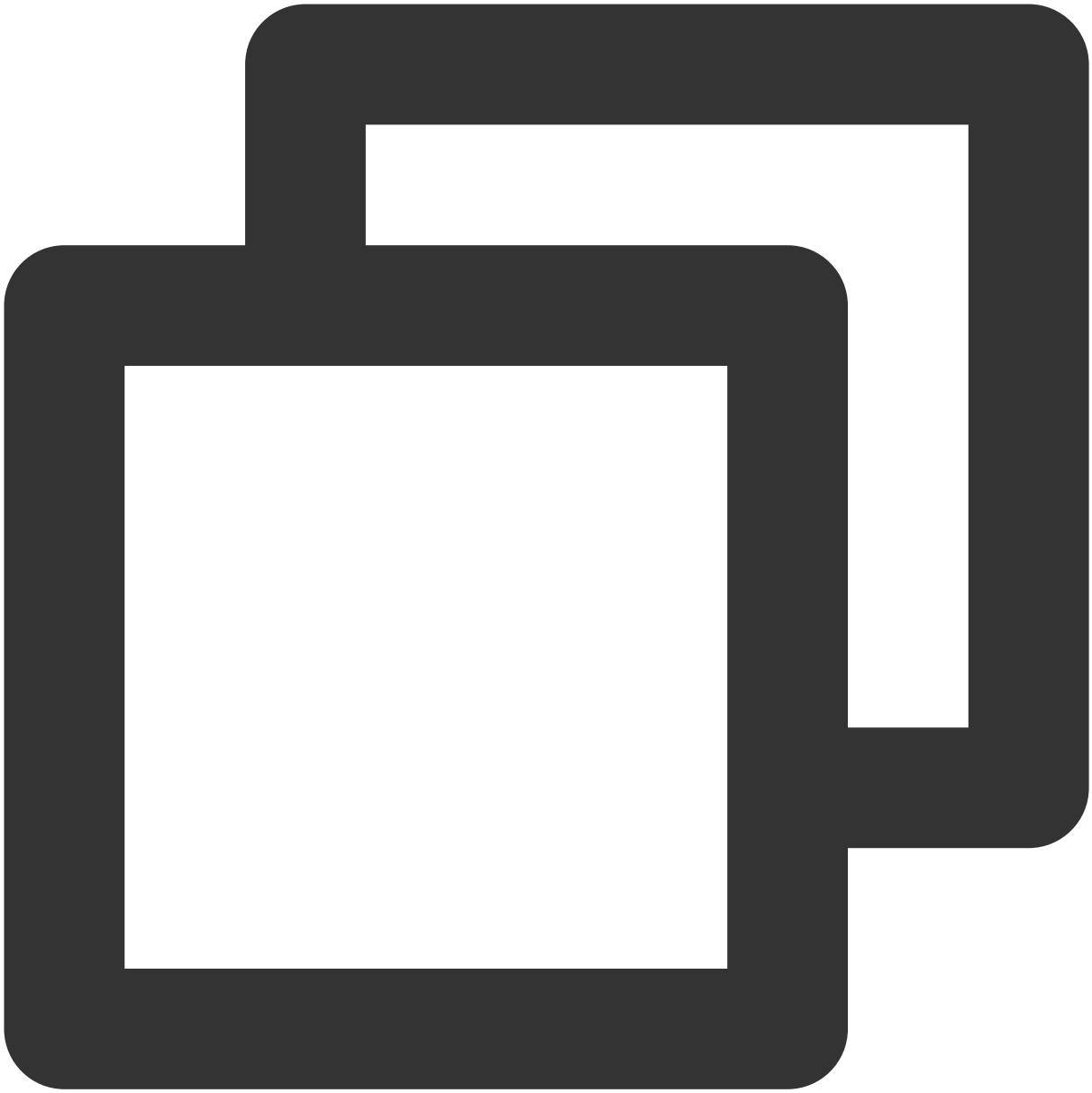


```
import io.opentelemetry.api.GlobalOpenTelemetry;
import io.opentelemetry.api.OpenTelemetry;
import io.opentelemetry.api.trace.Tracer;

public class AcquireTracerDemo {
    public void acquireTracer() {
        // The scope is used for defining the Event Tracking range. In most cases,
        String scope = this.getClass().getName();
        OpenTelemetry openTelemetry = GlobalOpenTelemetry.get();
        Tracer tracer = openTelemetry.getTracer(scope);
```

```
        }
    }
```

Performing Event Tracking for Business Methods





```
import io.opentelemetry.api.trace.Span;
import io.opentelemetry.api.trace.StatusCode;
import io.opentelemetry.api.trace.Tracer;
import io.opentelemetry.context.Scope;

// The Trace object can be obtained within business methods or passed into the busi
public void doTask(Tracer tracer) {
    // Create a Span.
    Span span = tracer.spanBuilder("doTask").startSpan();
    // Add some Attributes to the Span.
```

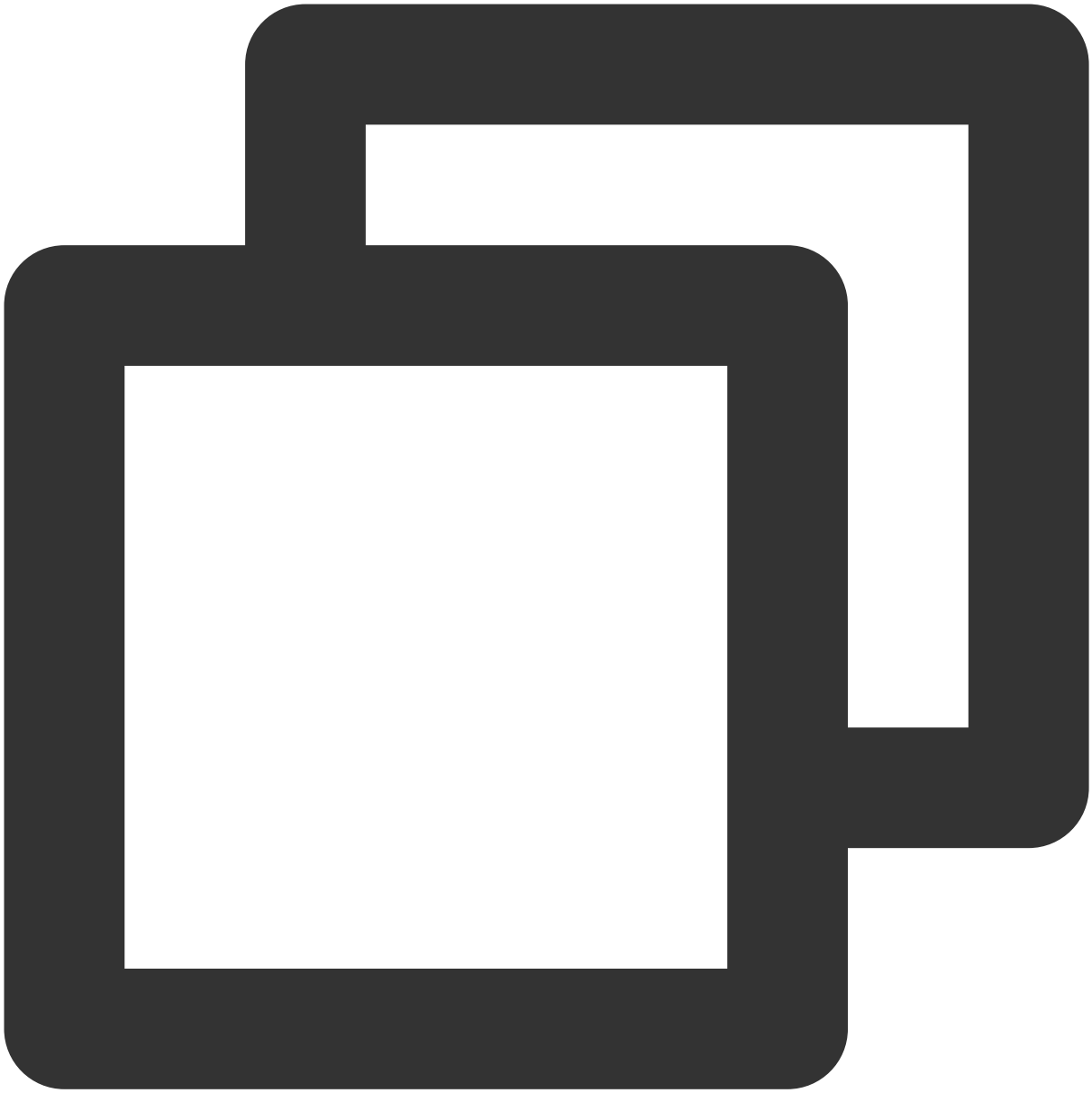
```
        span.setAttribute("RequestId", "5fc92ff1-8ca8-45f4-8013-24b4b5257666");
        // Set this Span as the current Span
        try (Scope scope = span.makeCurrent()) {
            doSubTask1();
            doSubTask2();
        } catch (Throwable t) {
            // Handle the exception. The exception information will be recorded in the
            span.recordException(t);
            span.setStatus(StatusCode.ERROR);
            throw t;
        } finally {
            // End the Span
            span.end();
        }
    }
```

Viewing the Results of Custom Event Tracking

In APM > call query, find the related call chain, and click the Span ID to enter the **End-to-End Details** page, where you can find the newly added Span through custom Event Tracking.

Custom Instance Name (Optional)

When multiple application processes connect APM using the same application name, they are displayed in APM as multiple instances under the same application. In most scenarios, the IP address can serve as a unique identifier for the application instance. However, if there are duplicate IP addresses in the system, other unique identifiers need to be used to define the instance name. For example, if the application in the system is launched directly through Docker, without being deployed on Kubernetes, there might be cases of duplicate container IP addresses. Users can set the instance name in the form of `host IP + container IP` .
See the following script and add the `host.name` field to the JVM startup parameters required for connecting APM with `-Dotel.resource.attributes` .



```
-Dotel.resource.attributes=service.name=my_service,token=my_demo_token,host.name=10
```

Reporting over SkyWalking Protocol

Last updated：2023-12-25 16:02:06

This document describes how to report the data of a Java application over the SkyWalking protocol.

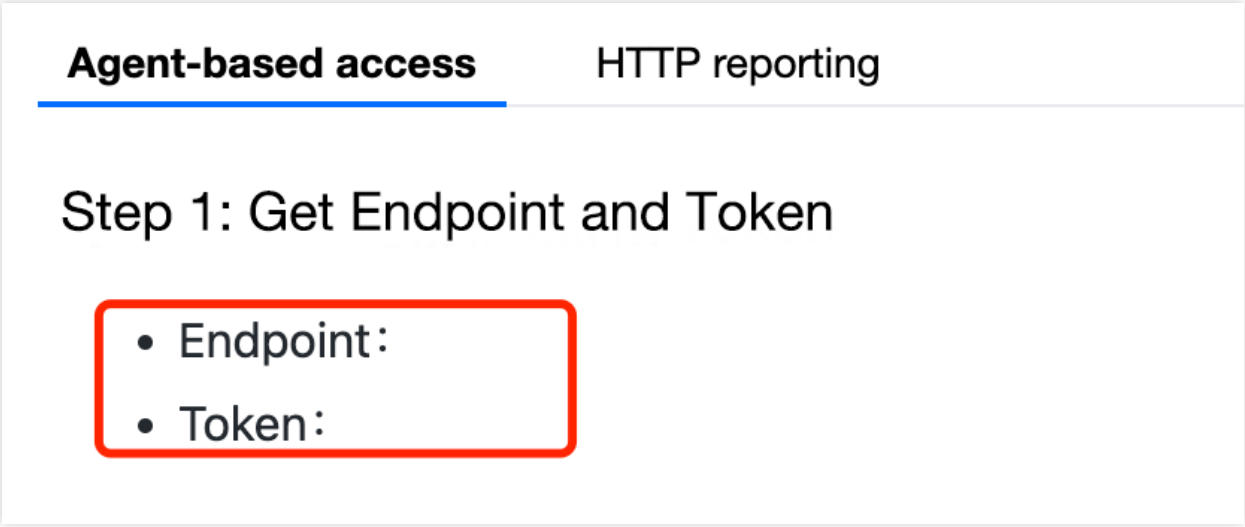
Prerequisites

Download SkyWalking 8.5.0 or later and place the extracted agent folder to a directory accessible to the Java process.
Plugins are in the `/plugins` directory. Put a new plugin in this directory during the startup phase to enable it, or remove it from this directory to disable it. Log files are output to the `/logs` directory by default.

Access Steps

Step 1. Get the endpoint and token

Log in to the APM console, enter the **Application monitoring** > **Application list** page, click **Access application**, and select the Java language and the SkyWalking data collection method.
Then, get the endpoint and token in the step of access method selection.

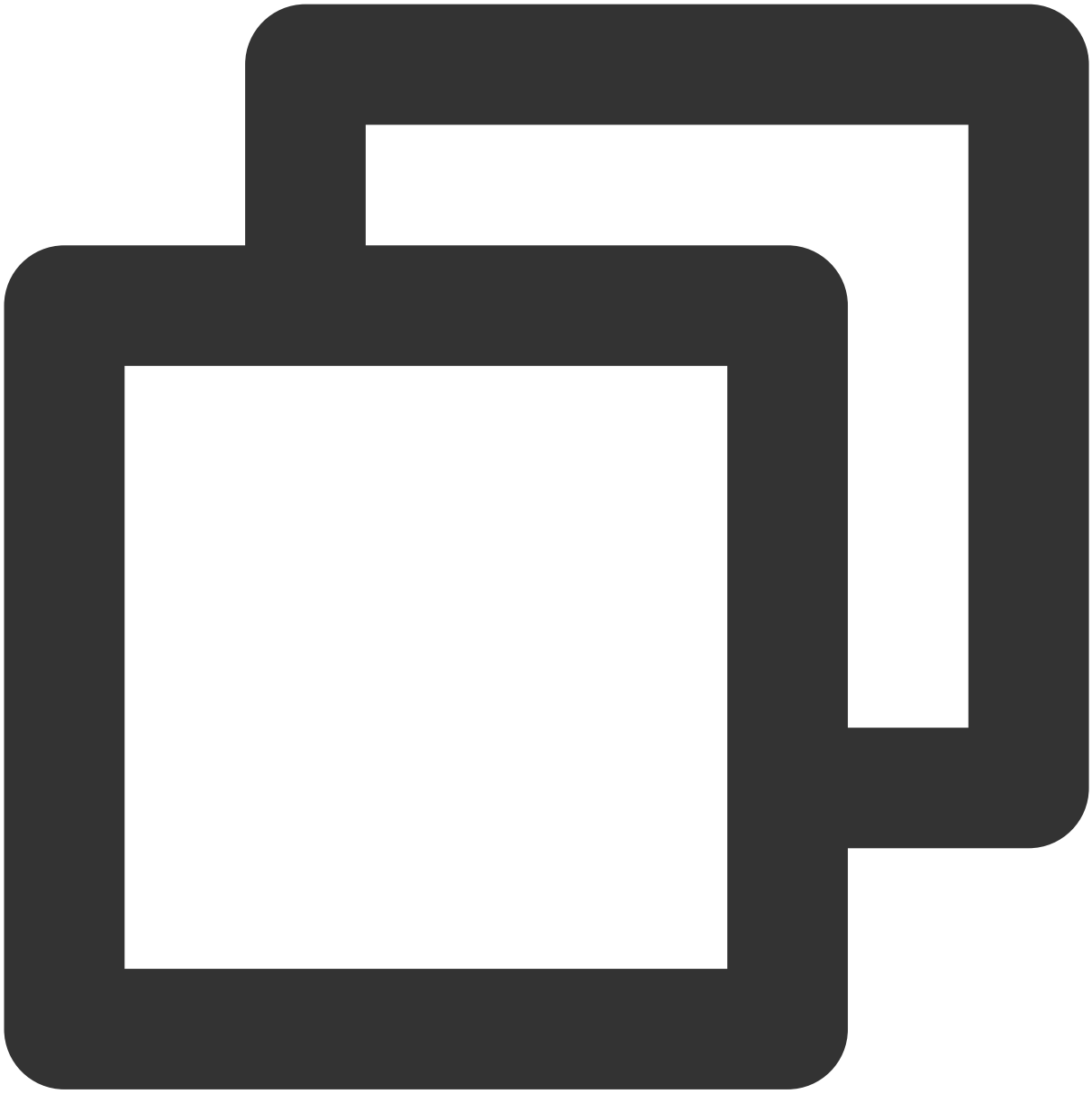


Step 2. Download Skywalking

If you have already used SkyWalking, skip this step.
Otherwise, download the latest version as instructed in Prerequisites.

Step 3. Configure parameters and names

Open the `agent/config/agent.config` file to configure the endpoint, token, and custom service name.



```
collector.backend_service=<endpoint>
agent.authentication=<Token>
agent.service_name=<reporting service name>
```

Note:

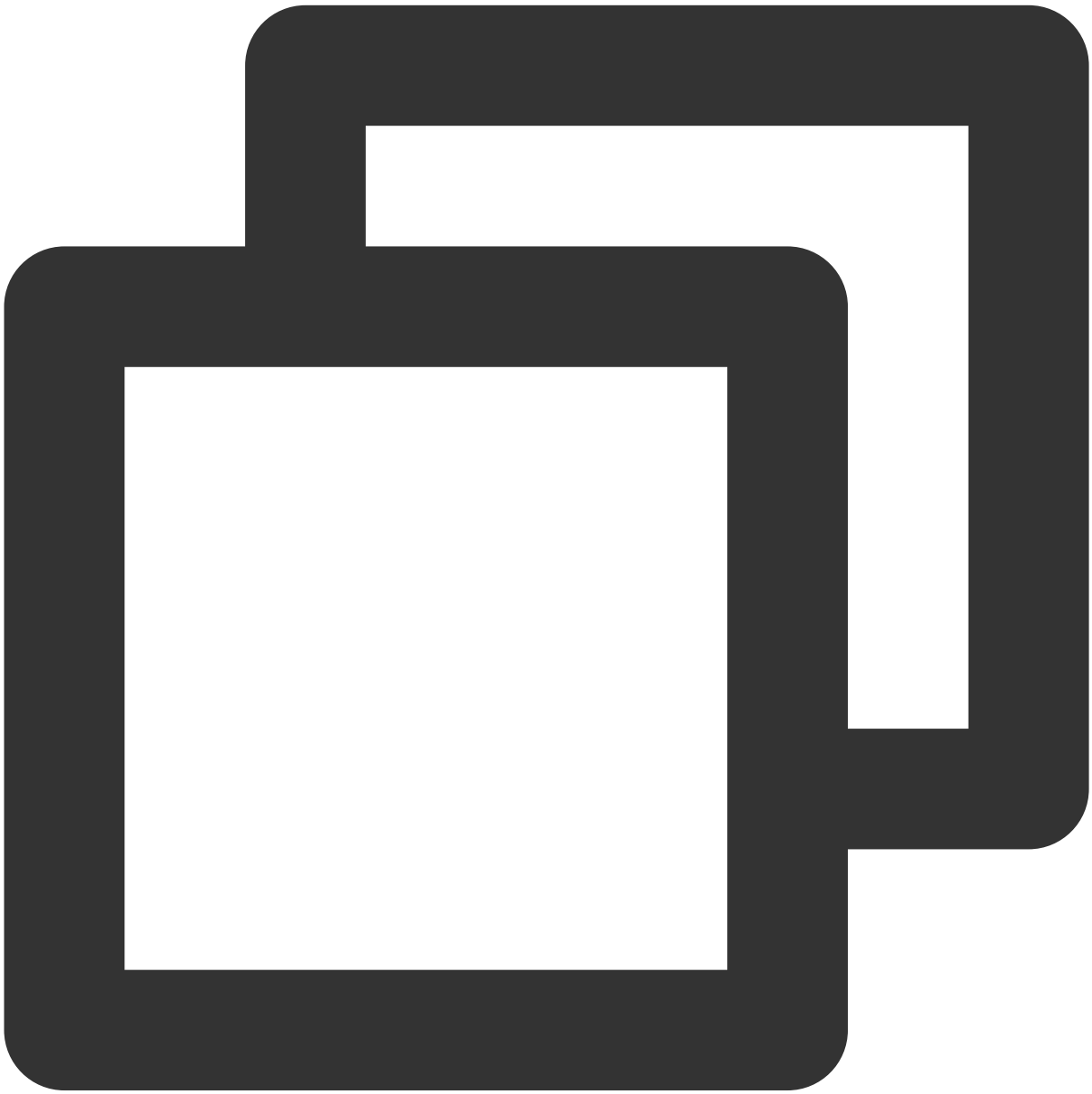
After modifying `agent.config`, remove the `#` before configuration items; otherwise, the changed information will not take effect.

Step 4. Specify the plugin path

Select an appropriate method based on the runtime environment of your application to specify the path of the SkyWalking agent.

Linux Tomcat 7/Tomcat 8

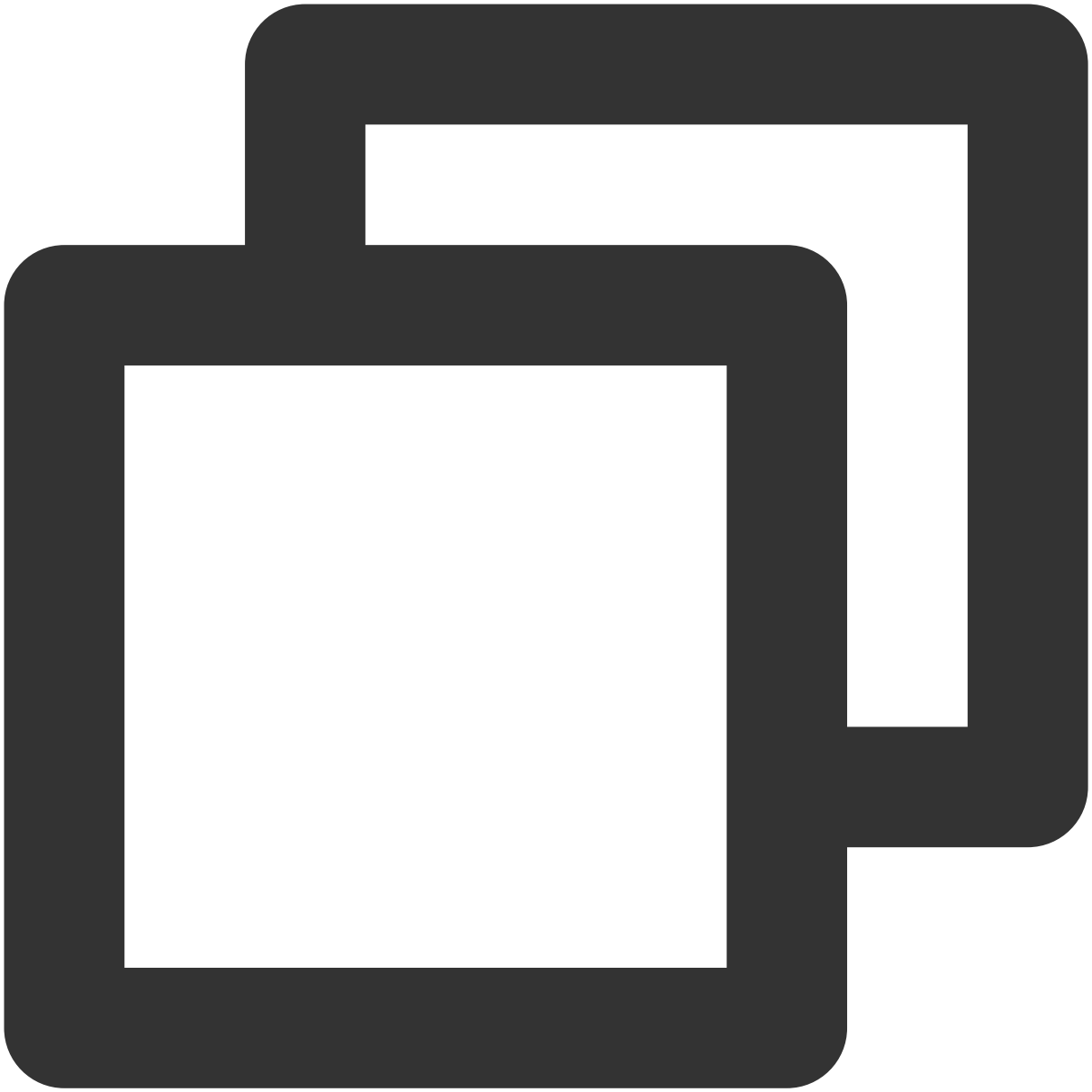
Add the following to the first line in `tomcat/bin/catalina.sh` :



```
CATALINA_OPTS="$CATALINA_OPTS -javaagent:<skywalking-agent-path>"; export CATALINA_
```

JAR file or Spring Boot

Add the `-javaagent` parameter to the startup command line of the application with the following content:



```
java -javaagent:<skywalking-agent-path> -jar yourApp.jar
```

Step 5. Restart the application

After completing the above deployment steps, restart the application as instructed in Install javaagent FAQs.

Accessing Python Application
Automatic Connecting Python Application for the TKE Environment (Recommended)

Last updated：2024-06-19 16:31:30

For Python applications deployed on TKE, APM offers automatic connection schemes, enabling automatic agent injection after the application is deployed to TKE, facilitating quick connection.
Automatic connecting Python applications for the TKE environment will use the community OpenTelemetry-Python scheme for agent injection. For more information on OpenTelemetry-Python, see the community OpenTelemetry-Python project.

Prerequisites

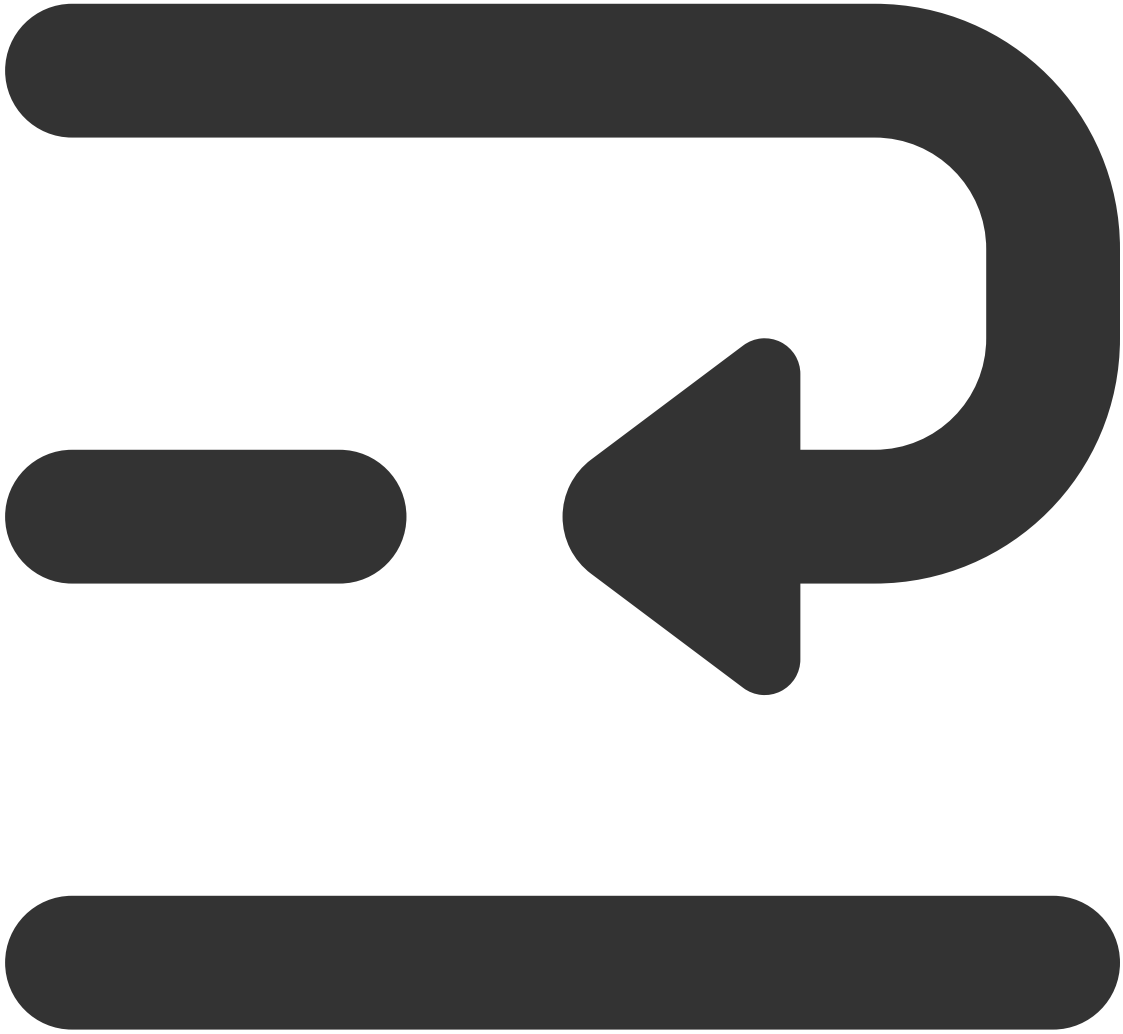
See Supported Components and Frameworks by the OpenTelemetry-Python Scheme to ensure that your Python versions, dependency libraries, and frameworks are within the supported range of the agent. For dependency libraries and frameworks supported by automatic instrumentation, data reporting is completed upon successful connection without the need to modify the code. If automatic Event Tracking does not meet your scenario, or you need to add business layer tracing, use the OpenTelemetry API for Custom Event Tracking.

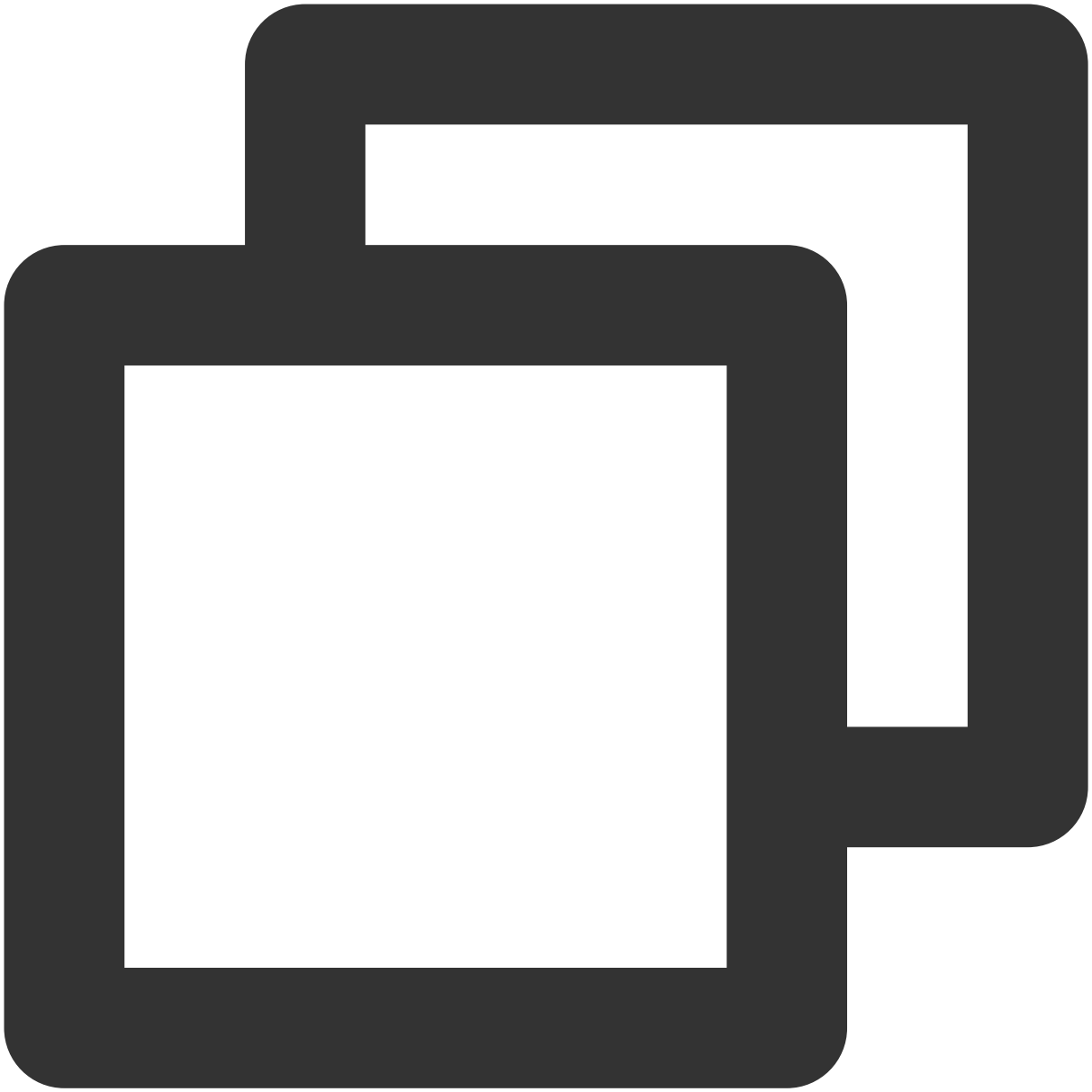
Step 1: Install Operator.

Install Operator in the TKE cluster, it's recommended to install Operator with one click on the APM console, for details see installing tencent-opentelemetry-operator.

Step 2: Add annotation to workload.

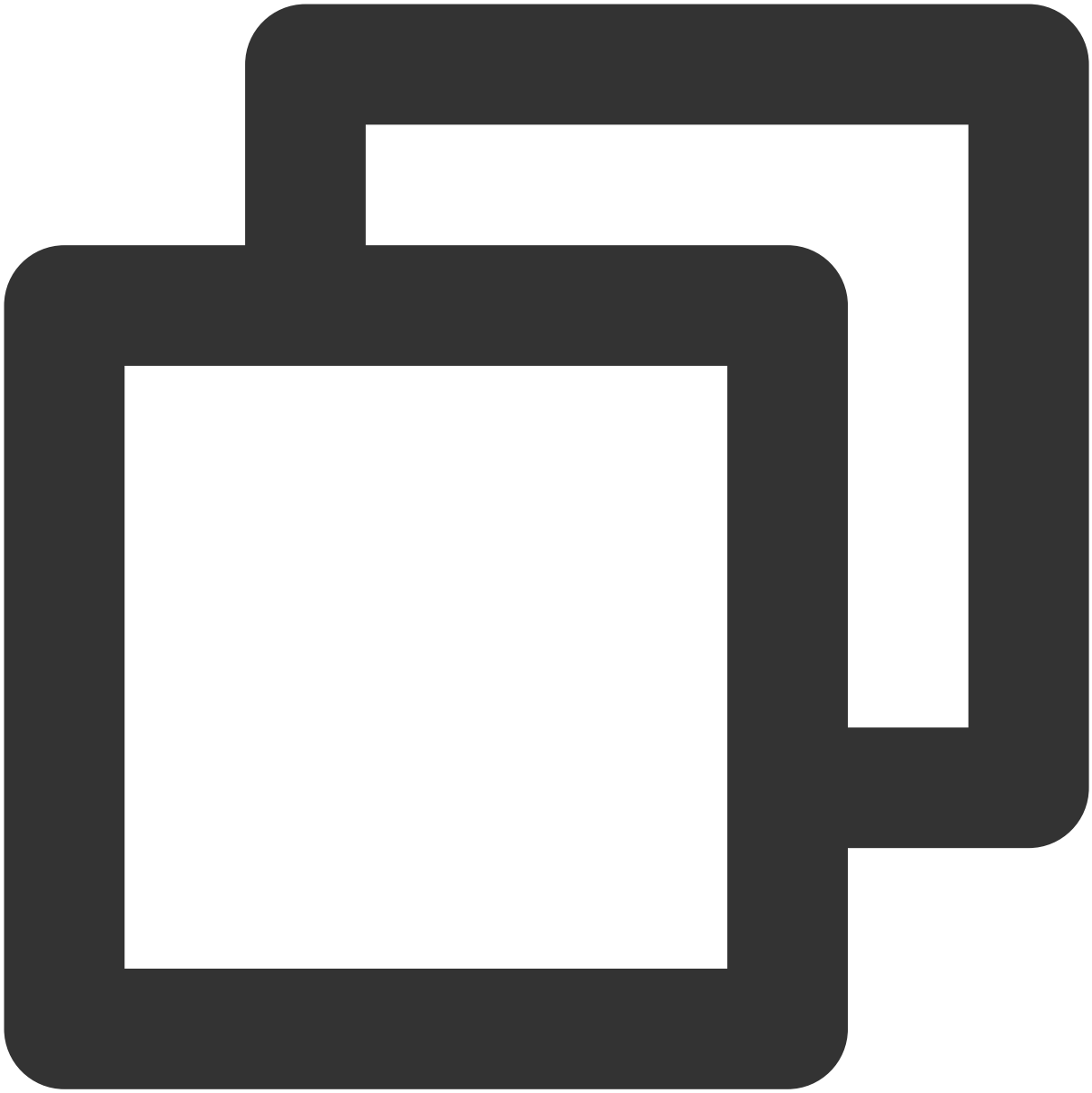
1. Log in to TKE console.
2. Click **Cluster** to enter the corresponding TKE cluster.
3. In **Workload**, locate the application that needs to connect with APM, click **More**, then click **Edit YAML**.
4. In the Pod annotation where the application is located, add the following content, then click **Complete** to finish the connection.





```
cloud.tencent.com/inject-python: "true"
cloud.tencent.com/otel-service-name: my-app  # Application name, processes that con
# The application name can be up to 63 characters and can only contain lowercase le
```

Note that this content needs to be added to `spec.template.metadata.annotations`, affecting the Pod's annotation, not the workload's annotation. You can see the following code snippet:



```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    k8s-app: my-app
  name: my-app
  namespace: default
spec:
  selector:
    matchLabels:
      k8s-app: my-app
```

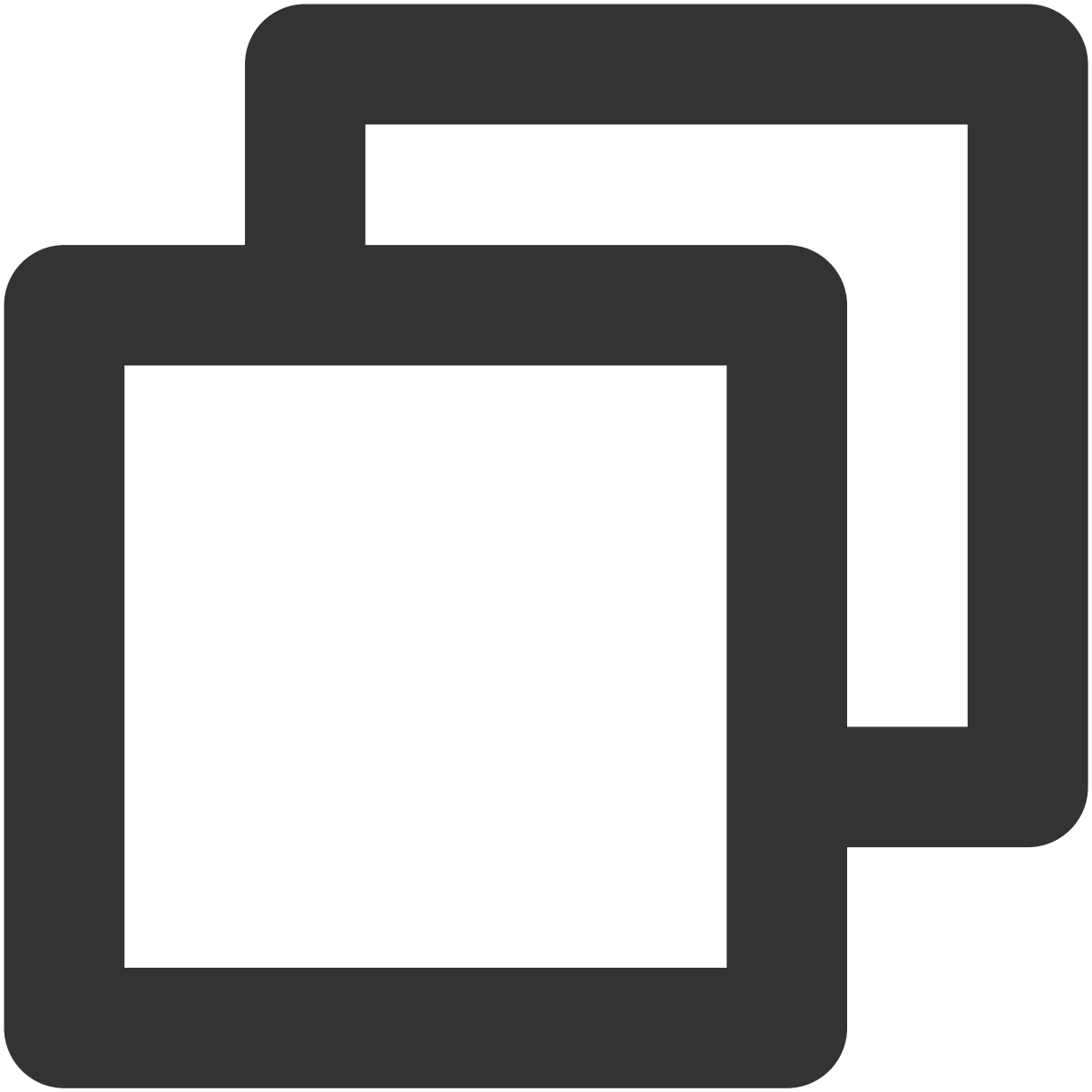
```
  template:
    metadata:
      labels:
        k8s-app: my-app
      annotations:
        cloud.tencent.com/inject-python: "true"  # Add it here.
        cloud.tencent.com/otel-service-name: my-app
    spec:
      containers:
        image: my-app:0.1
        name: my-app
```

Connection Verification

After an annotation is added to the workload, depending on the published policy, you can trigger the restart of the application Pod. The newly started Pod will automatically inject an agent and connect to the APM server to report monitoring data. The reported business system is the default business system of the Operator. In normal traffic cases, the connected applications will be displayed in APM > Application monitoring > Application list , and the connected application instances will be displayed in APM > Application monitoring > App details > **Instance monitoring**. Since there is a certain latency in the processing of observable data, if the applications or instances are not found in the console after connecting, wait about 30 seconds.

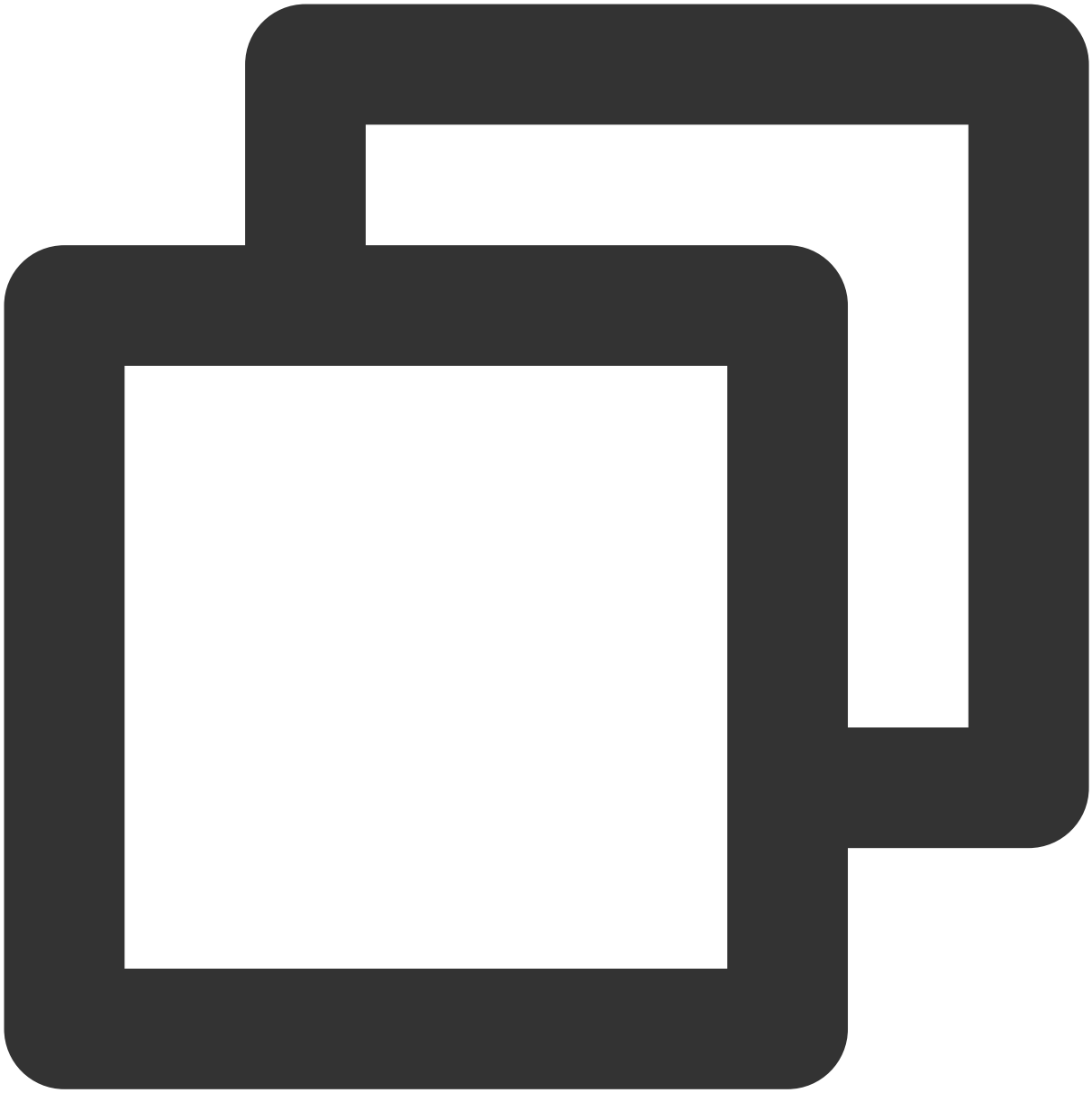
Django Application Precautions

If your application uses the Django framework, pay attention to the following points before connecting:
1. It is recommended to deploy the service using uWSGI. For deployment methods, see through uWSGI Hosting Django Application. Starting directly through the python command may cause reporting failures.
2. The introduction of OpenTelemetry-Python may result in Django applications no longer using the default configuration file. It is necessary to re-specify the configuration file through environment variables:



```
export DJANGO_SETTINGS_MODULE=mysite.settings
```

Or add environment variables through a YAML file:



```
env:
- name: DJANGO_SETTINGS_MODULE
  value: mysite.settings
```

More Connection Configuration Items (Optional)

At the Workload level, you can add more annotations to adjust the connect behaviors:

Configuration Item	Description
cloud.tencent.com/apm-	Specify the Token for the APM business system. If this configuration item is not

token	added, the Operator's configuration is used (corresponding to the Operator's `env.APM_TOKEN` field).
cloud.tencent.com/python-instr-version	Specify the Python agent version. If this configuration item is not added, the Operator's configuration is used (corresponding to the Operator's `env.PYTHON_INSTR_VERSION` field). The value can be `latest` (default) or a specific version number. For a list of specific version numbers, see Agent Version Information. It is not recommended to fill in this field unless necessary.

Connecting Python Applications Using OpenTelemetry-Python (Recommended)

Last updated：2024-06-19 16:31:30

Note:

OpenTelemetry is a collection of tools, APIs, and SDKs for monitoring, generating, collecting, and exporting telemetry data (metrics, logs, and traces) to help users analyze the performance and behavior of the software. For more information about OpenTelemetry, see the OpenTelemetry official website.

The OpenTelemetry community is active, with rapid technological changes, and widely compatible with mainstream programming languages, components, and frameworks, making its link-tracing capability highly popular for cloud-native microservices and container architectures.

This document will introduce how to connect Python applications using the OpenTelemetry-Python scheme provided by the community.

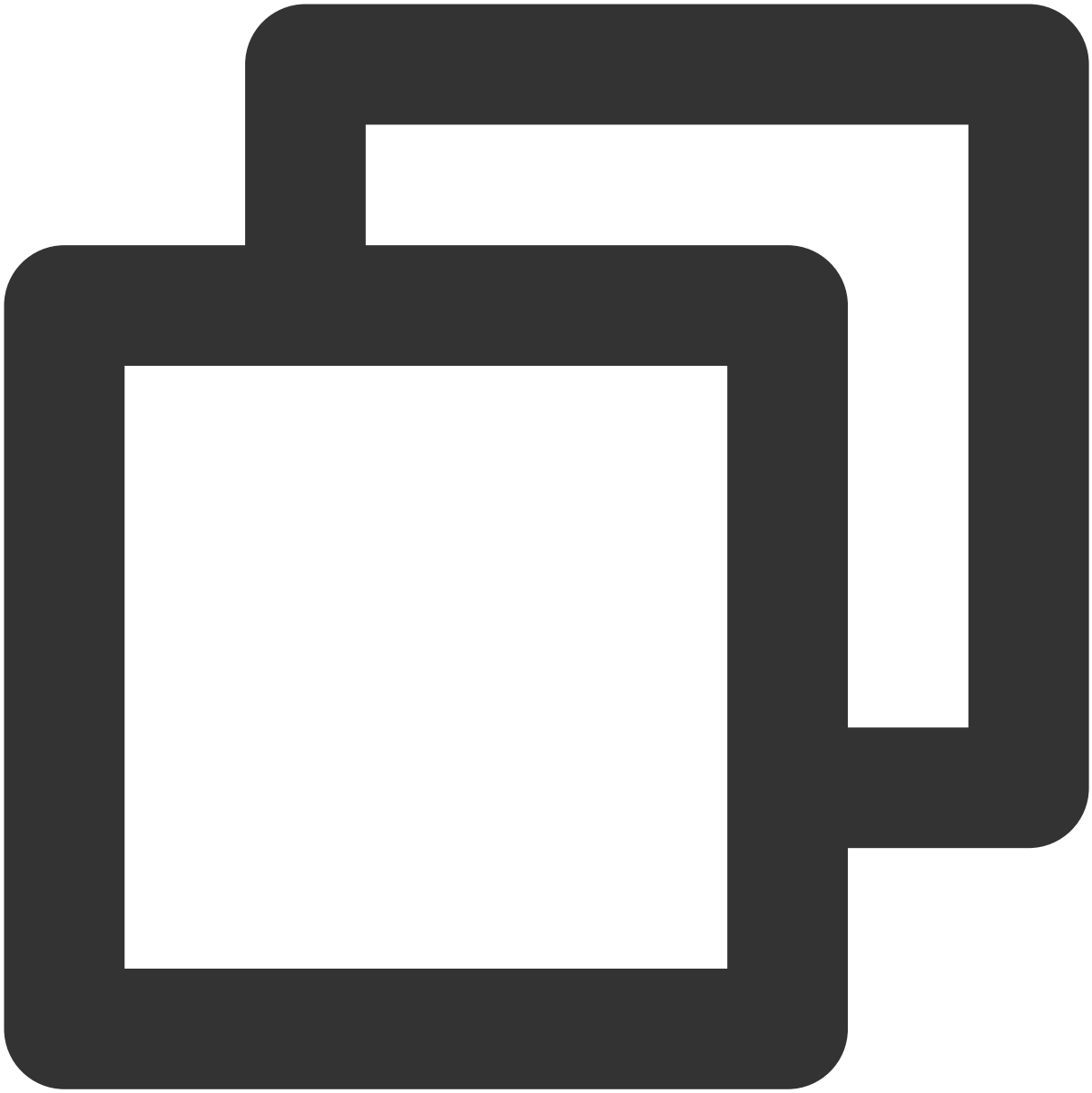
The OpenTelemetry-Python scheme provides automatic Event Tracking for commonly used dependency libraries and frameworks in the Python ecosystem, including Flask, Django, FastAPI, MySQL Connector, etc., enabling link information reporting without needing to modify the code. For other dependency libraries and frameworks that support automatic Event Tracking, see the complete list provided by the OpenTelemetry community.

Prerequisites

This scheme supports Python 3.6 and above.

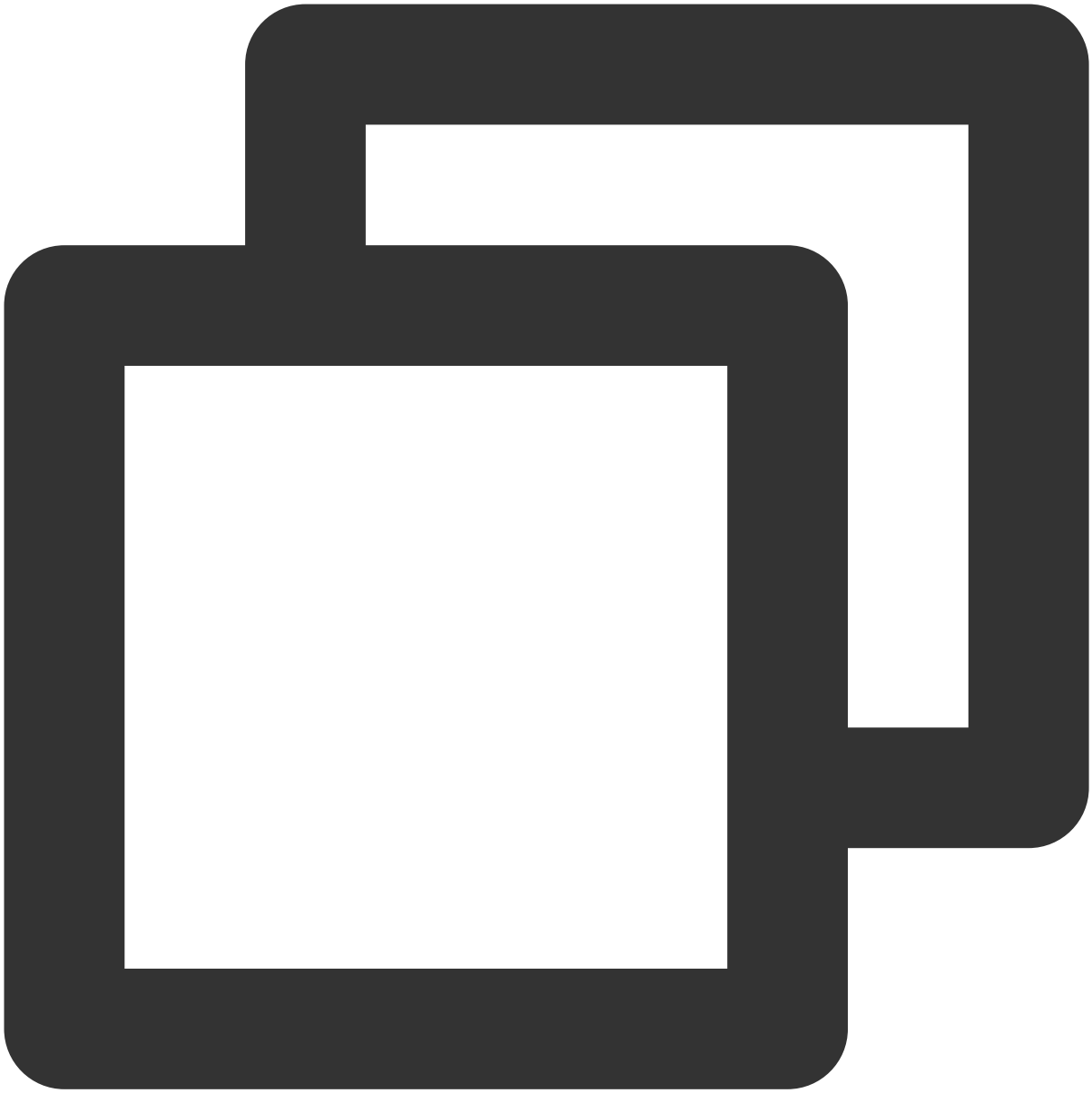
Demo

Required dependencies are as follows:



```
pip install flask
pip install mysql-connector-python
pip install redis
pip install requests
```

The demo code app.py provides 3 HTTP APIs through the Flask framework. Set up the corresponding MySQL and Redis services yourself or directly purchase Cloud Services.



```
from flask import Flask
import requests
import time
import mysql.connector
import redis

backend_addr = 'https://example.com/'

app = Flask(__name__)

# Accessing External Site
```

```
@app.route('/')
def index():
    start = time.time()
    r = requests.get(backend_addr)
    return r

# Accessing Database
@app.route('/mysql')
def func_rdb():
    cnx = mysql.connector.connect(host='127.0.0.1', database="<DB-NAME>", user='<DB
    cursor = cnx.cursor()
    val = "null"
    cursor.execute("select value from table_demo where id=1;")
    val = cursor.fetchone()[0]
    cursor.close()
    cnx.close()
    return "rdb res:" + val

# Accessing Redis
@app.route("/redis")
def func_kvop():
    client = redis.StrictRedis(host="localhost", port=6379)
    val = "null"
    val = client.get('foo').decode("utf8")
    return "kv res:" + val

app.run(host='0.0.0.0', port=8080)
```

Preliminary steps: Get the connect point and Token.

1. Log in to the TCOP console.
2. In the left menu column, select **Application Performance Management > Application monitoring**, and click **Application list > Access application**.
3. In the **Data access** drawer frame that pops up on the right, click the **Python** language.
4. On the **Access Python application** page, select the **region** and **Business System** you want to connect.
5. Select **Access protocol type** as **OpenTelemetry**.
6. **Reporting method** Choose your desired reporting method, and obtain your **Access Point** and **Token**.

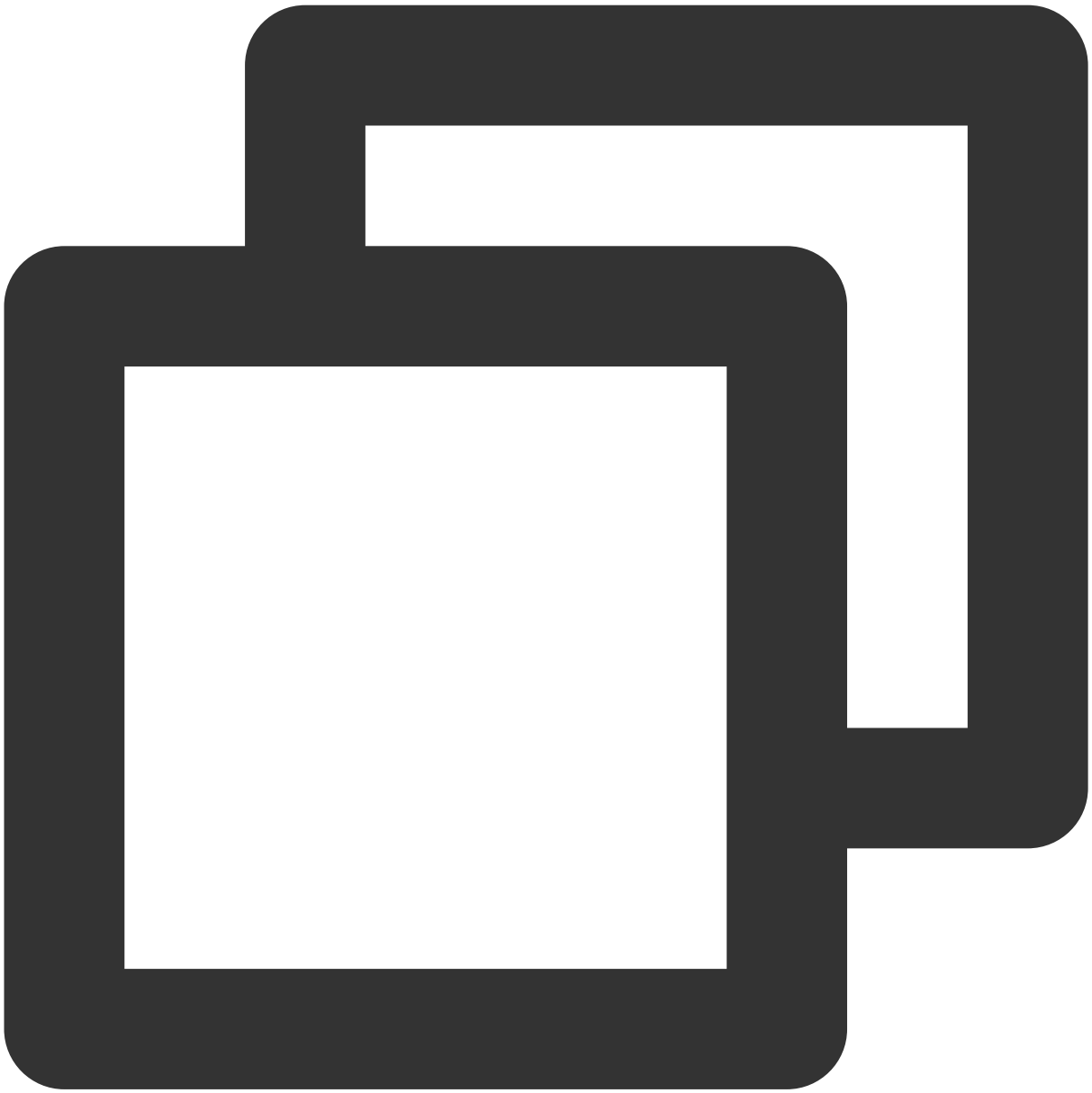
Note:

Private network reporting: Using this reporting method, your service needs to run in the Tencent Cloud VPC. Through VPC connecting directly, you can avoid the security risks of public network communication and save on reporting traffic overhead.

Public network reporting: If your service is deployed locally or in non-Tencent Cloud VPC, you can report data in this method. However, it involves security risks in public network communication and incurs reporting traffic fees.

Connecting Python Applications

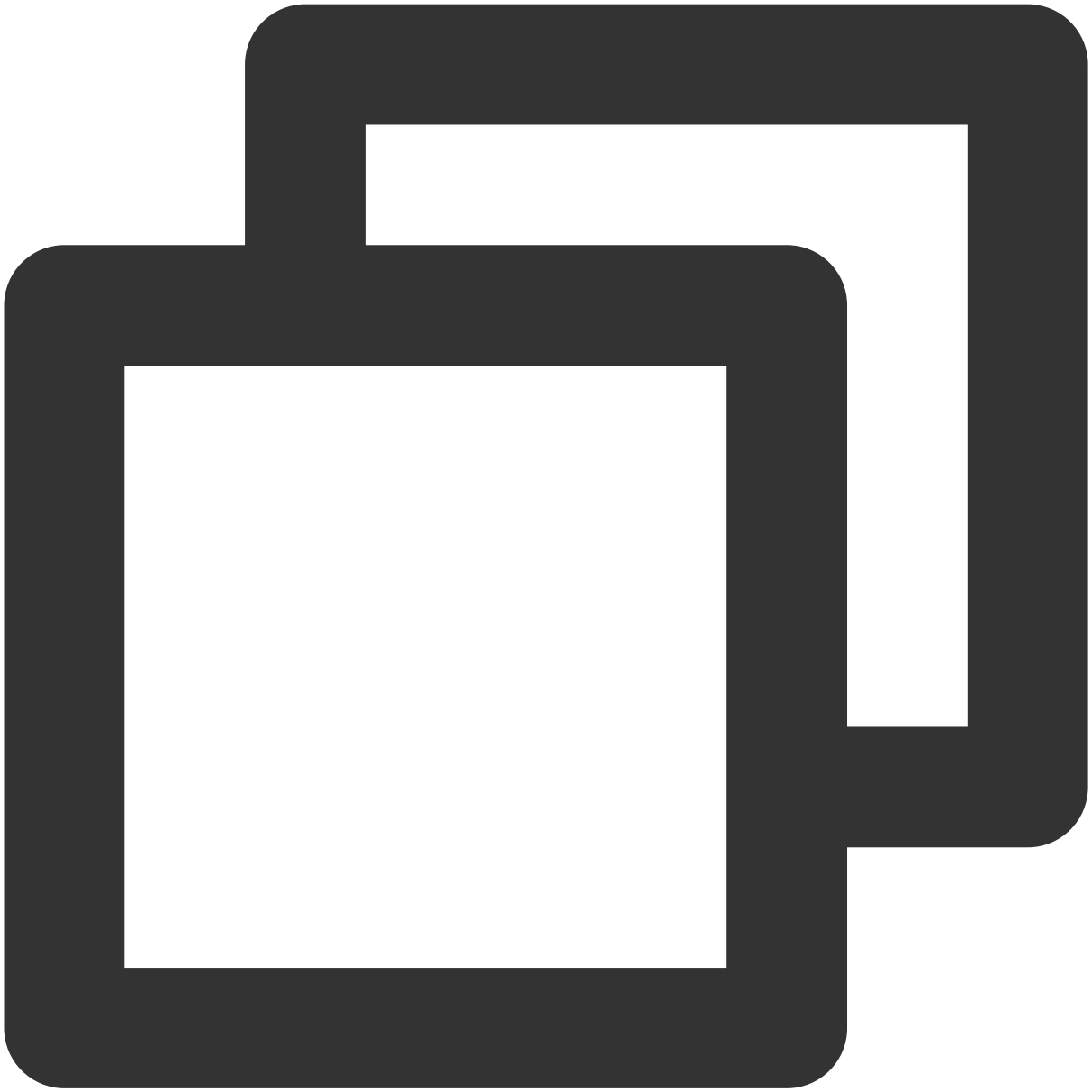
Step 1: Install the required dependency packets.



```
pip install opentelemetry-instrumentation-redis
pip install opentelemetry-instrumentation-mysql
pip install opentelemetry-distro opentelemetry-exporter-otlp

opentelemetry-bootstrap -a install
```

Step 2: Add runtime parameters.

Start the Python application with the following command:



```
opentelemetry-instrument \\
--traces_exporter otlp_proto_grpc \\
--metrics_exporter none \\
--service_name <serviceName> \\
--resource_attributes token=<token>,host.name=<hostName> \\
--exporter_otlp_endpoint <endpoint> \\
python3 app.py
```

The corresponding field descriptions are as follows:

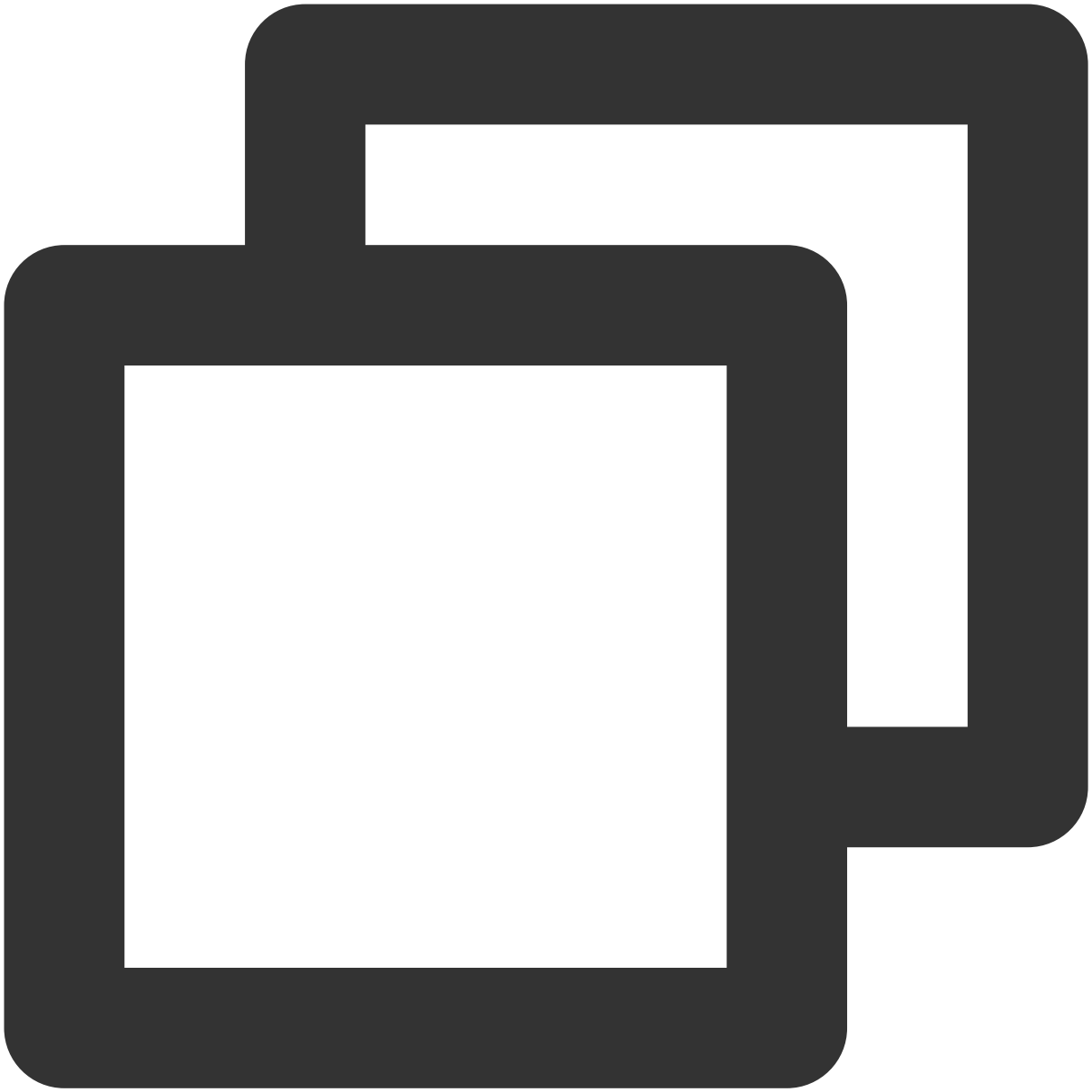
`<serviceName>` : Application name. Multiple application processes connecting with the same serviceName are displayed as multiple instances under the same application in APM. The application name can be up to 63 characters and can only contain lowercase letters, digits, and the separator (-), and it must start with a lowercase letter and end with a digit or lowercase letter.

`<token>` : The business system Token obtained in the preliminary steps.

`<hostName>` : The hostname of this instance, which is the unique identifier of the application instance. It can usually be set to the IP address of the application instance.

`<endpoint>` : The connect point obtained in the preliminary steps.

The content below uses `myService` as the application name, `myToken` as the Business System Token, `192.168.0.10` as the hostname, and `https://pl-demo.ap-guangzhou.apm.tencentcs.com:4317` as the connect point example, the complete startup command is:



```
opentelemetry-instrument \\
--traces_exporter otlp_proto_grpc \\
--metrics_exporter none \\
--service_name myService \\
--resource_attributes token=myToken,host.name=192.168.0.10 \\
--exporter_otlp_endpoint https://pl-demo.ap-guangzhou.apm.tencentcs.com:4317/ \\
python3 app.py
```

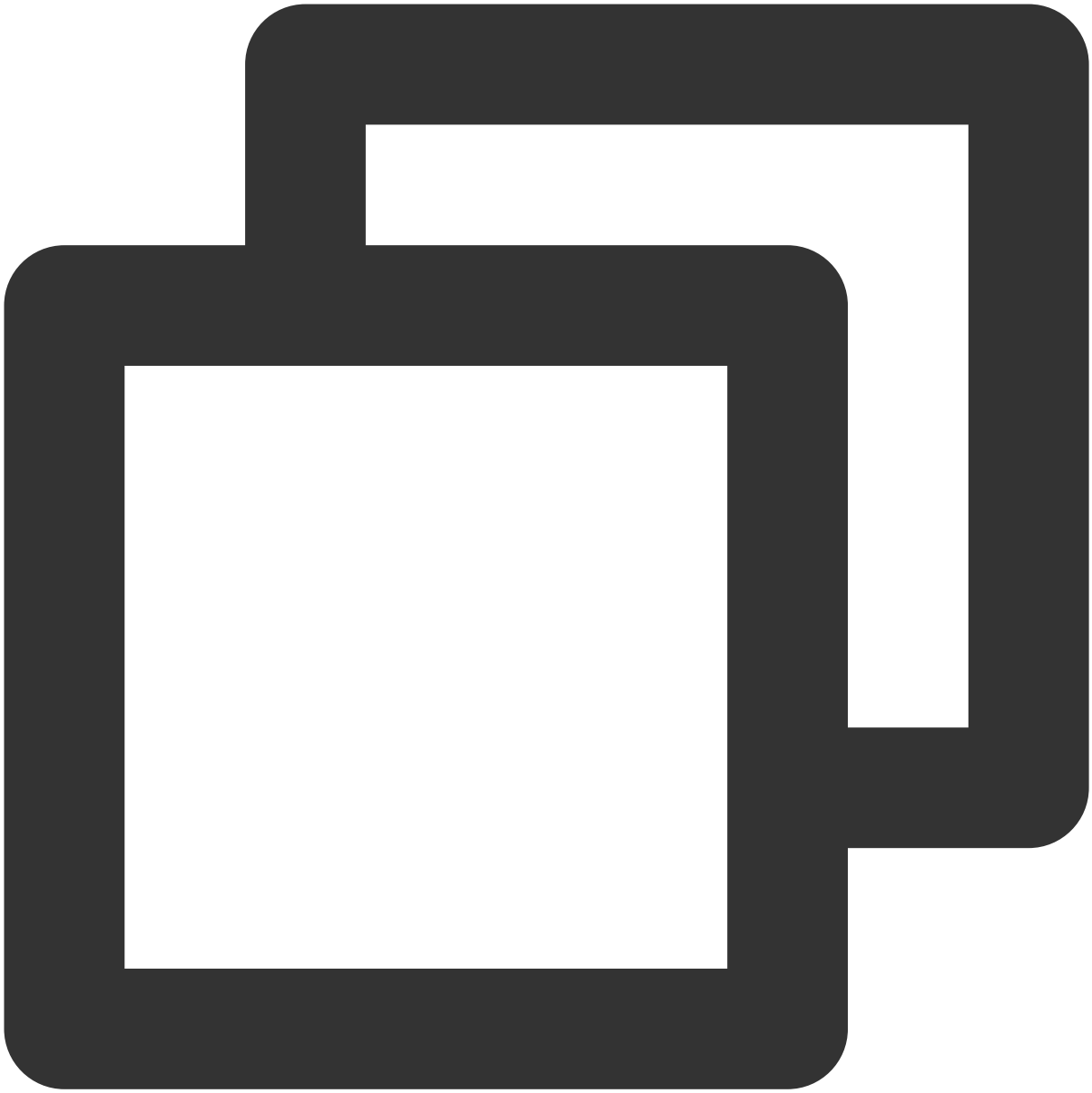
Connection Verification

After the Python application starts, access the corresponding API through port 8080, for example, `https://localhost:8080/` . If there is normal traffic, the connected application is displayed in APM > Application monitoring > Application list, and the connected application instance will be displayed in APM > Application monitoring > Application details > **Instance monitoring**. Since there is a certain delay in the processing of observable data, if the application or instance is not found on the console after connection, wait for about 30 seconds.

Django Application Precautions

If your application uses the Django framework, before you connect it through the OpenTelemetry-Python scheme, pay attention to the following items:

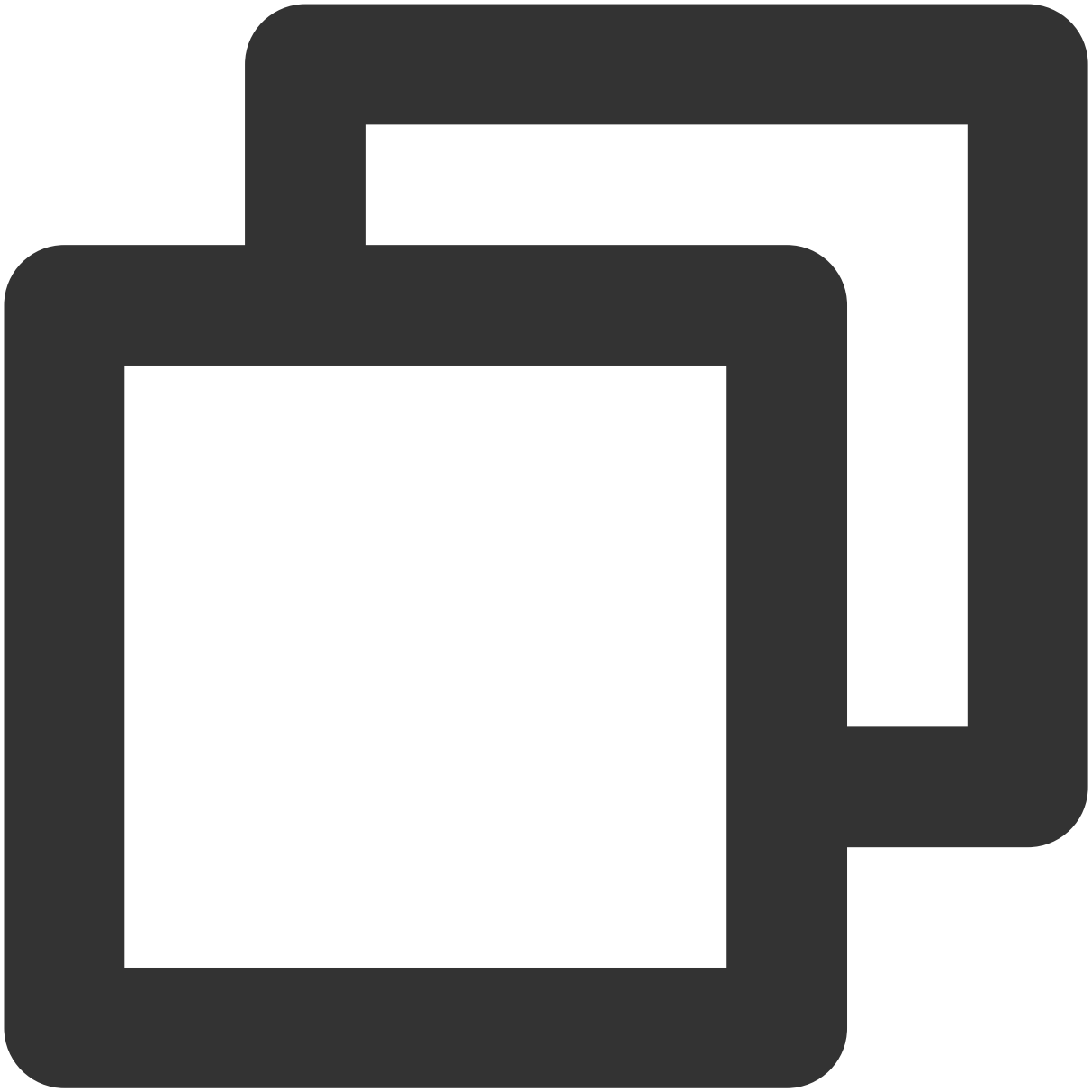
1. It is recommended to deploy the service using uWSGI. For deployment methods, see through uWSGI Hosting Django Application. Starting directly through the python command may cause reporting failures.
2. The introduction of OpenTelemetry-Python may result in Django applications no longer using the default configuration file. It is necessary to re-specify the configuration file through environment variables:



```
export DJANGO_SETTINGS_MODULE=mysite.settings
```

Custom Event Tracking (Optional)

When automatic Event Tracking does not meet your scenario, or you need to add business layer Event Tracking, you can see the content below and use the OpenTelemetry API to add custom Event Tracking. This document only shows the most basic custom Event Tracking method. The OpenTelemetry community offers more flexible custom Event Tracking. For specific methods of use, you can see the Python Custom Event Tracking Documentation provided by the OpenTelemetry community.



```
from opentelemetry import trace
import requests

backend_addr = 'https://example.com/'
app = Flask(__name__)

@app.route('/')
def index():
   r = requests.get(backend_addr) # For external requests initiated by requests.get
    slow() # Call a custom function
    return r
```

```
def slow():
    tracer = trace.get_tracer(__name__)
    # Custom functions are not within the automatic Event Tracking range of OpenTel
    with tracer.start_as_current_span("child_span")
        time.sleep(5)
        return
```

Reporting over Jaeger Protocol

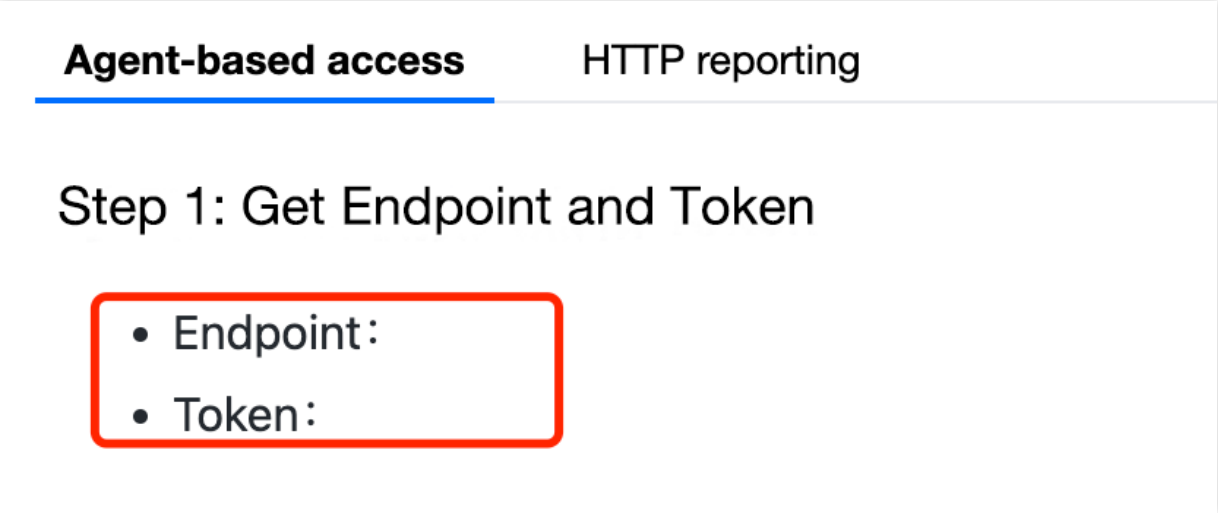
Last updated：2023-12-25 16:02:37

This document describes how to report the data of a Python application over the Jaeger protocol.

Directions

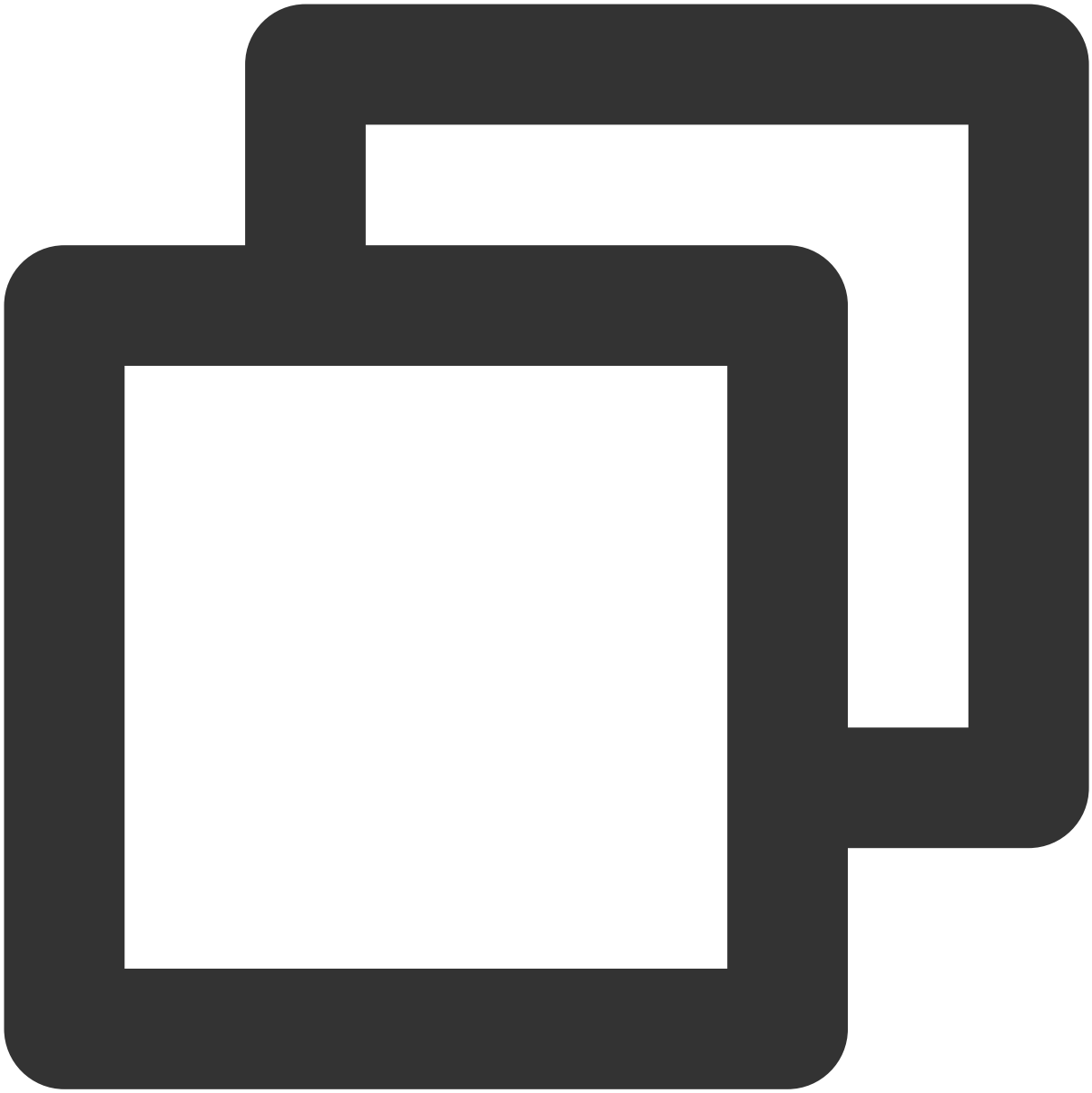
Step 1. Get the endpoint and token

Log in to the APM console, enter the **Application monitoring** > **Application list** page, click **Access application**, and select the Python language and Jaeger data collection method. Then, get the endpoint and token in the step of access method selection.



Step 2. Install the Jaeger agent

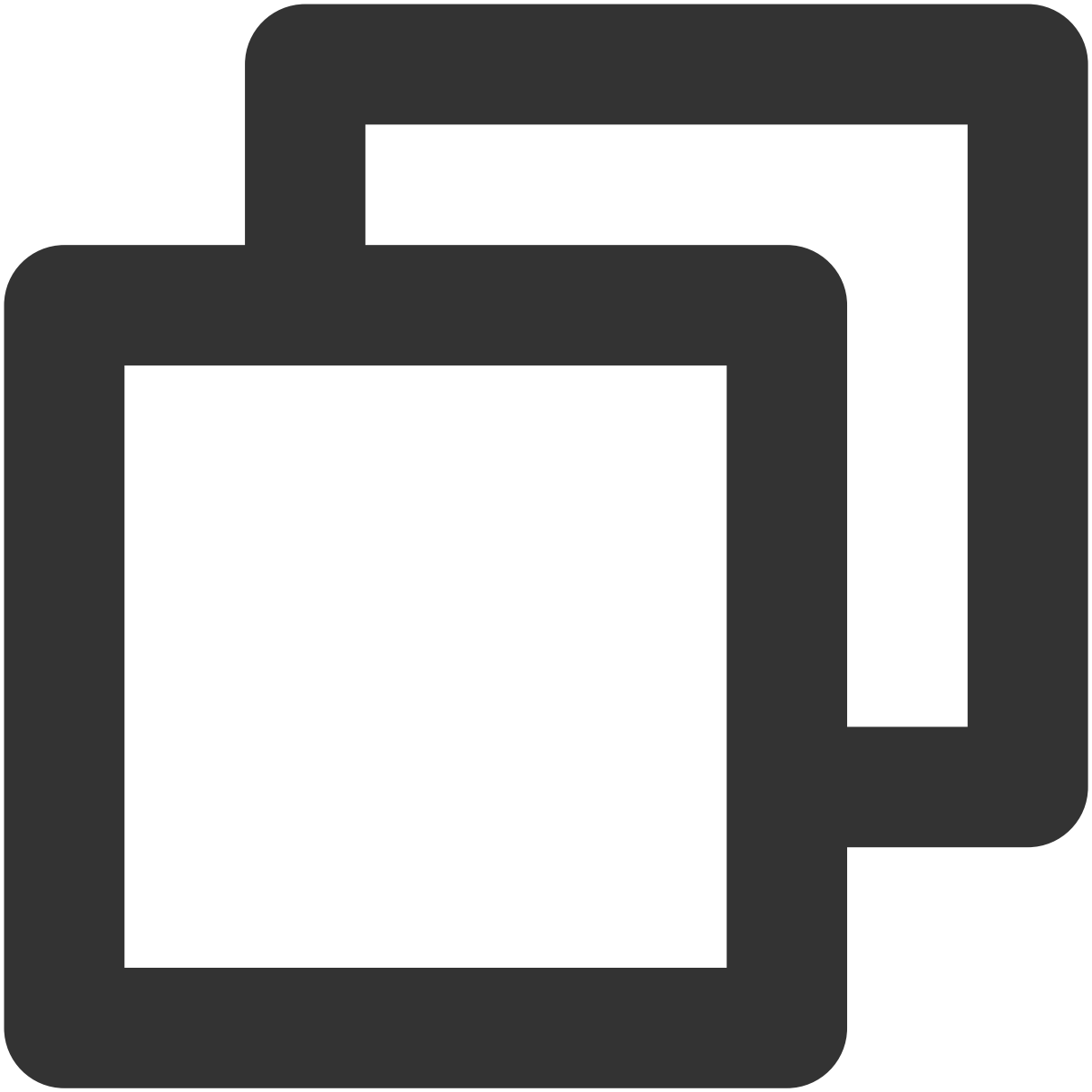
1. Download the Jaeger agent.
2. Run the following command to start the agent:



```
nohup ./jaeger-agent --reporter.grpc.host-port={{endpoint}} --jaeger.tags=token={{t
```

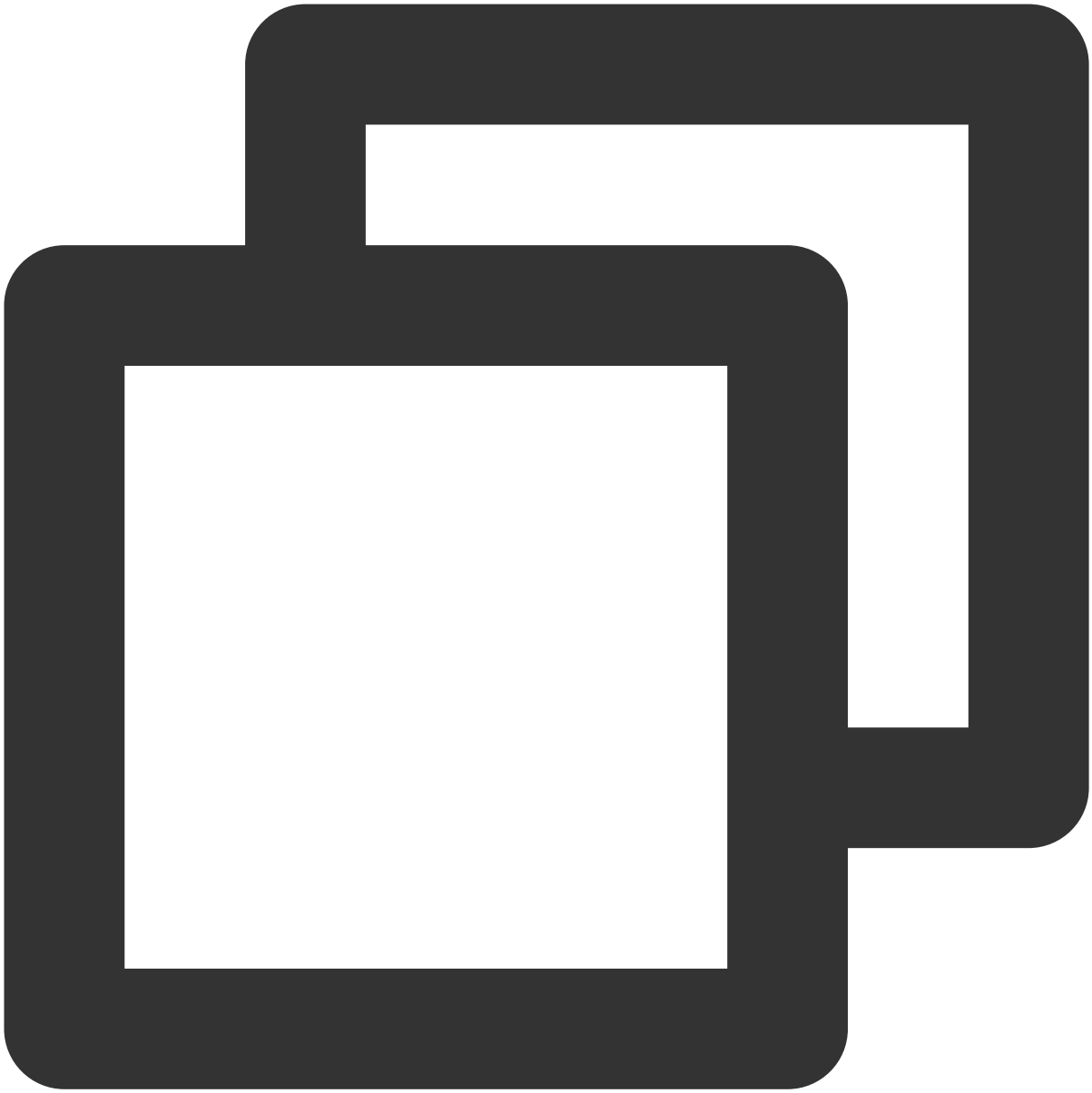
Step 3. Report data through Jaeger

1. Run the following command to install the `jaeger_client` package.



```
pip install jaeger_client
```

2. Create the following Python file and tracer object to trace all requests.



```
from jaeger_client import Config
import time
from os import getenv

# Configure the address of the Jaeger agent, which is the localhost by default.
JAEGER_HOST = getenv('JAEGER_HOST', 'localhost')
SERVICE_NAME = getenv('JAEGER_HOST', 'my_service_test')


def build_your_span(tracer):
    with tracer.start_span('yourTestSpan') as span:
```

```
        span.log_kv({'event': 'test your message', 'life': 42})
        span.set_tag("span.kind", "server")
        return span


def build_your_tracer():
    my_config = Config(
        config={
            'sampler': {
                'type': 'const',
                'param': 1,
            },
            'local_agent': {
                'reporting_host': JAEGER_HOST,
                'reporting_port': 6831,
            },
            'logging': True,
        },
        service_name=SERVICE_NAME,
        validate=True
    )
    tracer = my_config.initialize_tracer()

    return tracer


if __name__ == "__main__":
    tracer = build_your_tracer()
    span = build_your_span(tracer)
    time.sleep(2)
    tracer.close()
```

Note:

Currently, Jaeger supports frameworks such as Flask, Django, and gRPC for data reporting. For more information, see the following documents:

jaeger-client-python

3rd-Party OpenTracing API Contributions

Accessing Node.js Application
Automatic Connecting Node.js Applications for TKE Environment (Recommended)

Last updated：2024-06-19 16:31:30

For Node.js applications deployed on TKE, APM provides an automatic connect scheme. This allows for the automatic injection of agents after the application is deployed to TKE, facilitating quick connection.
TKE environment automatically connected Node.js applications will use the community OpenTelemetry-JavaScript scheme for agent injection. For more information about OpenTelemetry-JavaScript, see the community OpenTelemetry-Javascript project.

Prerequisites

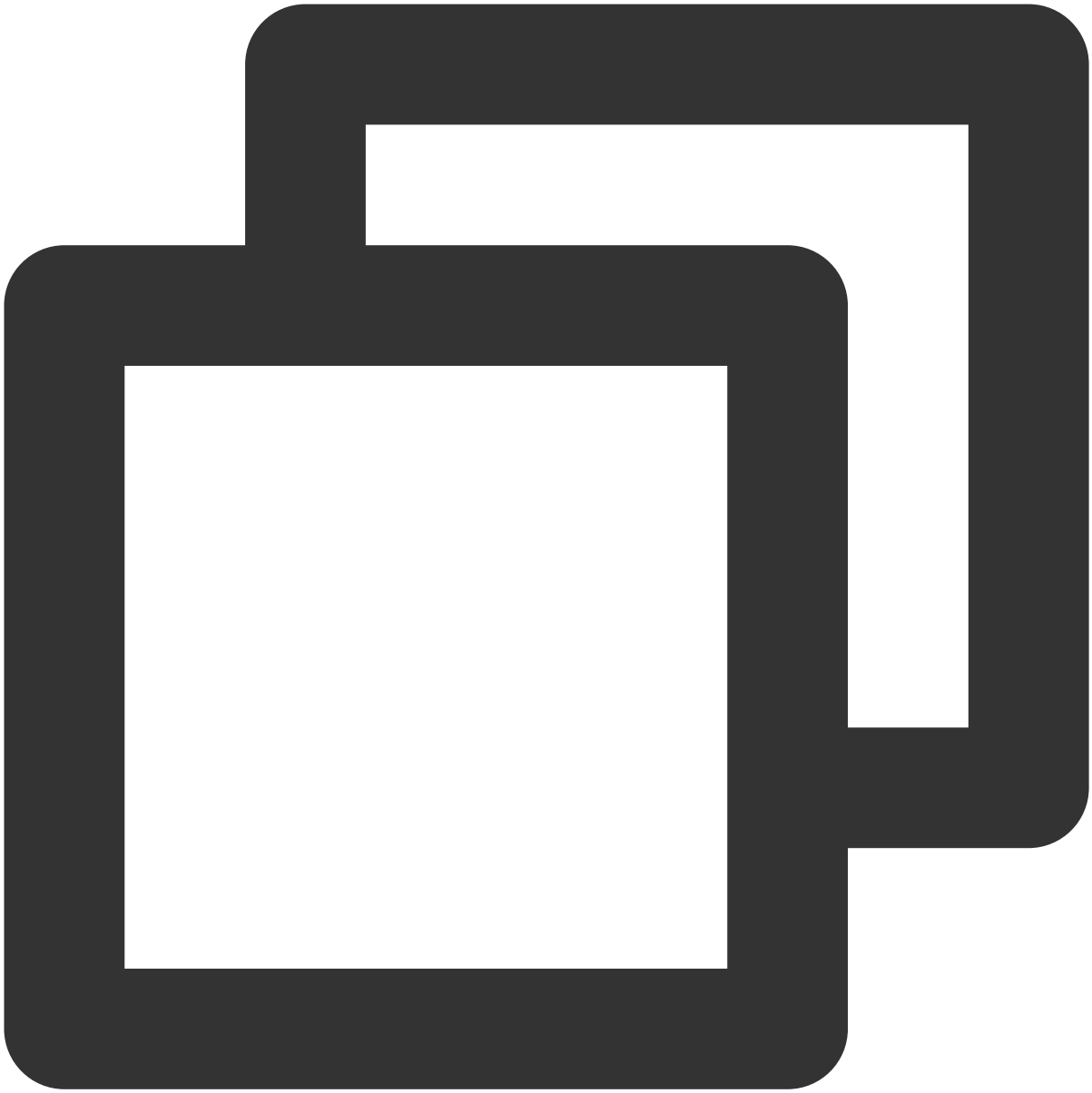
See OpenTelemetry-JavaScript scheme supported components and frameworks to ensure that the Node.js version, dependency libraries, and frameworks are within the supported range of the agent. For dependency libraries and frameworks supported by automatic Event Tracking, data reporting can be completed after a successful connection without modifying the code. If automatic Event Tracking does not meet your needs, or you need to add business layer instrumentation, use OpenTelemetry API for custom metrics.

Step 1: Install Operator.

Install Operator in the TKE cluster, it's recommended to install Operator with one click on the APM console, for details see installing tencent-opentelemetry-operator.

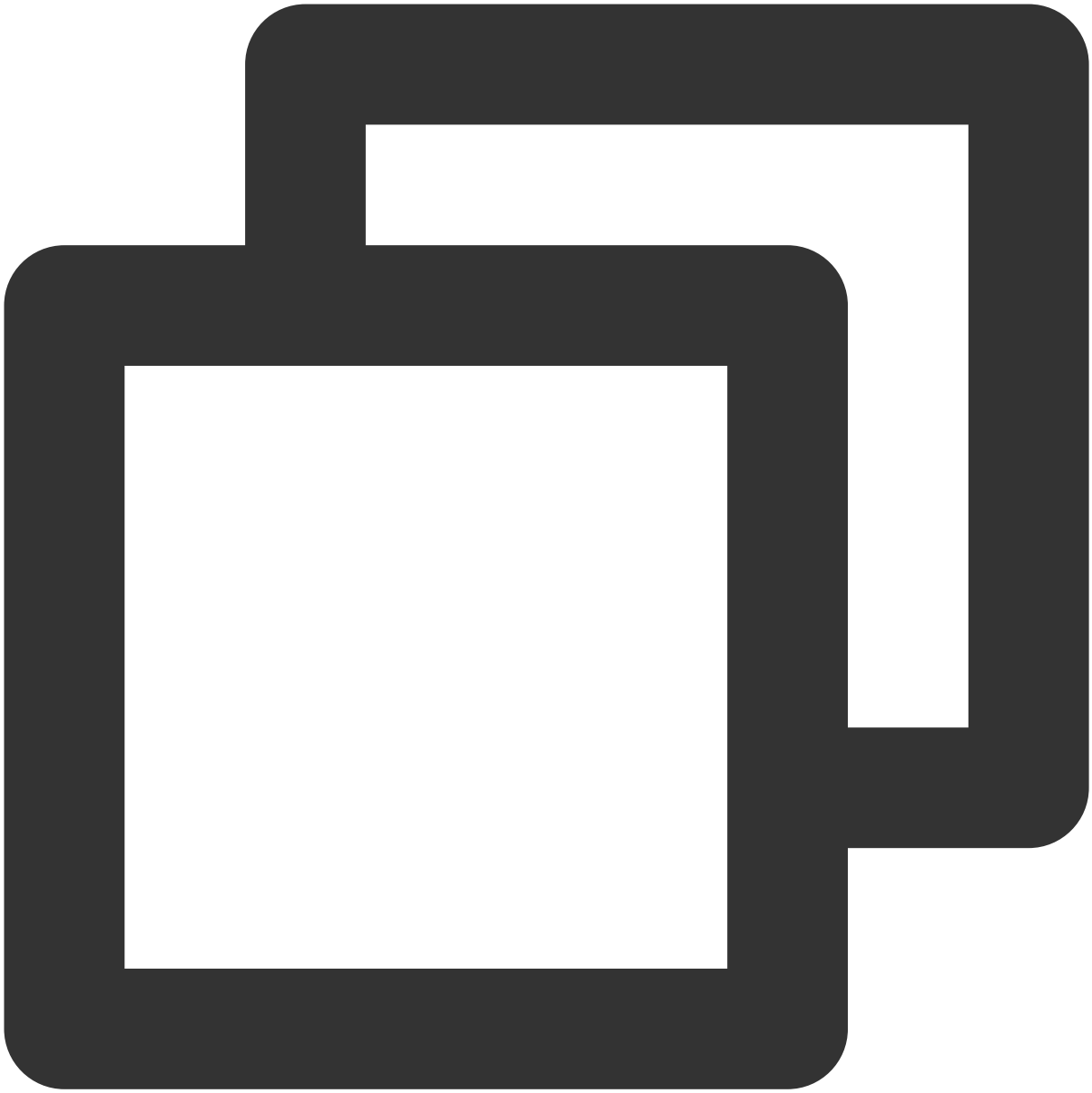
Step 2: Add annotation to workload.

1. Log in to TKE console.
2. Click **Cluster** to enter the corresponding TKE cluster.
3. In **Workload**, you can find the application that needs to connect APM, click **More**, then click **Edit YAML**.
4. Apply the following content in the Pod annotation, then click **Complete** to finish the connection.



```
cloud.tencent.com/inject-nodejs: "true"
cloud.tencent.com/otel-service-name: my-app  # Application name, processes that con
# The application name can be up to 63 characters and can only contain lowercase le
```

Note that this content needs to be added to `spec.template.metadata.annotations`, affecting the Pod's annotation, not the workload's annotation. You can see the following code snippet:



```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    k8s-app: my-app
  name: my-app
  namespace: default
spec:
  selector:
    matchLabels:
      k8s-app: my-app
```

```
  template:
    metadata:
      labels:
        k8s-app: my-app
      annotations:
        cloud.tencent.com/inject-nodejs: "true"  # Add here
        cloud.tencent.com/otel-service-name: my-app
    spec:
      containers:
        image: my-app:0.1
        name: my-app
```

Connection Verification

After an annotation is added to the workload, based on different publish policies, it triggers the restart of the application Pod. The newly started Pod will automatically inject an agent and connect to the APM server to report monitoring data, with the business system reported being the default for Operator. Under normal traffic conditions, the connected applications will be displayed in APM > Application monitoring > Application list , and the connected application instances will be displayed in APM > Application monitoring > App details > **Instance monitoring**. Due to a certain delay in processing observable data, if the application or instance is not found in the console immediately after connection, wait for about 30 seconds.

More Connection Configuration Items (Optional)

At the Workload level, you can add more annotations to adjust the connect behaviors:

Configuration Item	Description
cloud.tencent.com/apm-token	Specify the Token for the APM business system. If this configuration item is not added, the configuration of the Operator is used (corresponding to the Operator's `env.APM_TOKEN` field).
cloud.tencent.com/nodejs-instr-version	Specify the Node.js agent version. If this configuration item is not added, the Operator's configuration is used (corresponding to the Operator'snull `env.NODEJS_INSTR_VERSION` field). The values can be `latest` (default) or a specific version number. For a list of specific version numbers, see Agent Version Information. It is not recommended to fill in this field unless necessary.

Connecting Node.js Applications Using the OpenTelemetry-JS Scheme (Recommended)

Last updated：2024-06-19 16:31:30

Note:

OpenTelemetry is a collection of tools, APIs, and SDKs for monitoring, generating, collecting, and exporting telemetry data (metrics, logs, and traces) to help users analyze the performance and behavior of the software. For more information about OpenTelemetry, see the OpenTelemetry official website.

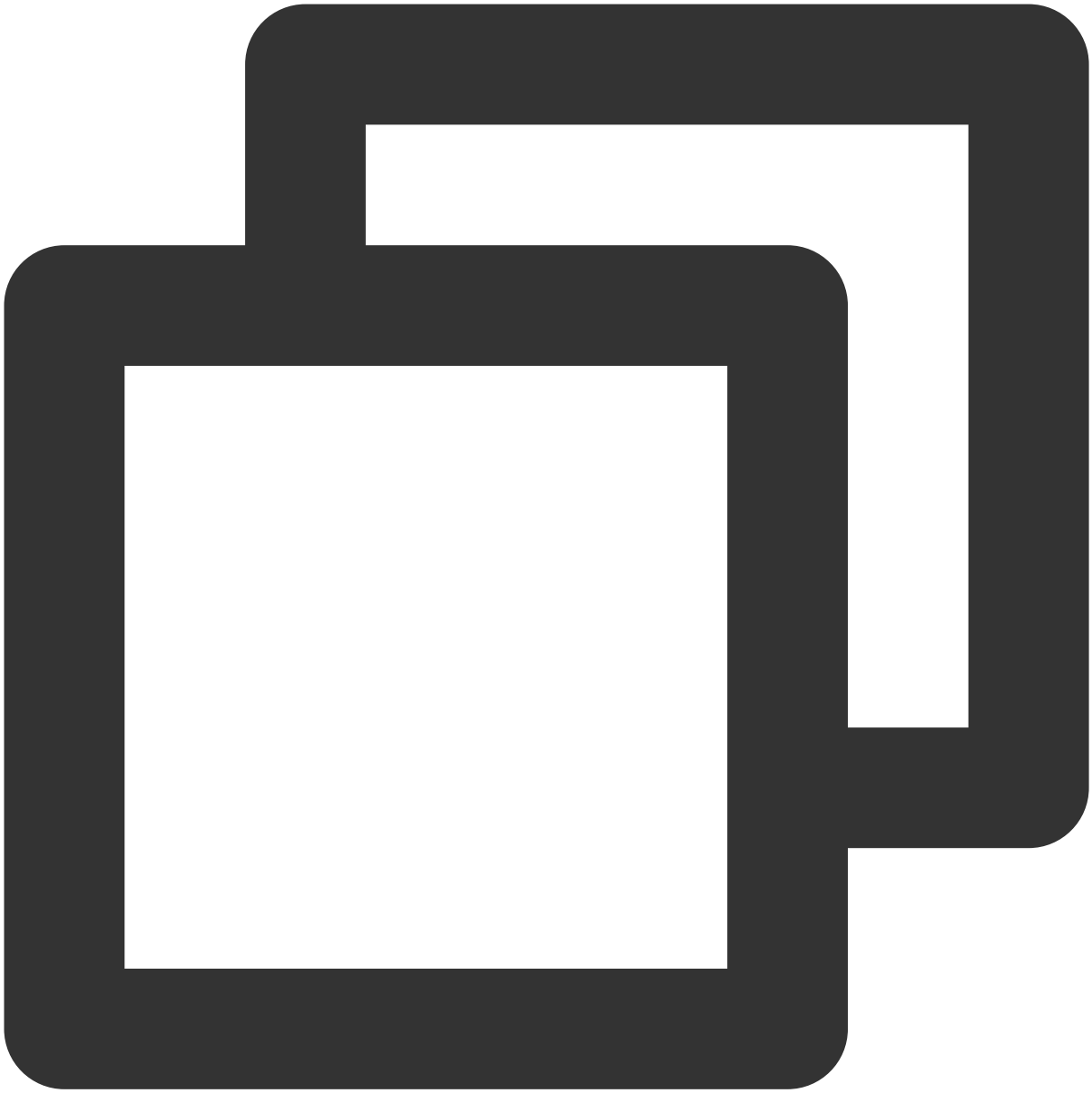
The OpenTelemetry community is active, with rapid technological changes, and widely compatible with mainstream programming languages, components, and frameworks, making its link-tracing capability highly popular for cloud-native microservices and container architectures.

This document will introduce how to connect Node.js applications using the OpenTelemetry-JS scheme through related operations.

The OpenTelemetry-JS scheme provides automatic Event Tracking for common Node.js modules and frameworks, including Express, mysql, gRPC, etc., enabling link information reporting without needing to modify the code. For other modules and frameworks that support automatic Event Tracking, see the complete list provided by the OpenTelemetry community.

Demo

The demo code main.js provides 3 HTTP APIs through Express. Set up the corresponding MySQL and Redis services yourself or directly purchase Cloud Services.



```
"use strict";

const axios = require("axios").default;
const express = require("express");
const redis = require('./utils/redis');
const dbHelper = require("./utils/db");
const app = express();

app.get("/remoteInvoke", async (req, res) => {
  const result = await axios.get("http://cloud.tencent.com");
  return res.status(200).send(result.data);
```

```
});

app.get("/redis", async(req, res) => {
  let queryRes = await redis.getKey("foo")
  res.json({ code: 200, result: queryRes})
})

app.get("/mysql", async(req, res) => {
  let select = select * from table_demo;
  await dbHelper.query(select);
  res.json({ code: 200, result: "mysql op ended"})
})

app.use(express.json());

app.listen(8080, () => {
  console.log("Listening on http://localhost:8080");
});
```

Preliminary steps: Get the connect point and Token.

1. Log in to the TCOP console.
2. In the left menu column, select **Application Performance Management > Application monitoring**, and click **Application list > Access application**.
3. In the **Data access** drawer frame that pops up on the right, click **Node** language.
4. On the **Access Node application** page, select the **region** and **business system** you want to connect.
5. Select **Access protocol type** as **OpenTelemetry**.
6. **Reporting method** Choose your desired reporting method, and obtain your **Access Point** and **Token**.

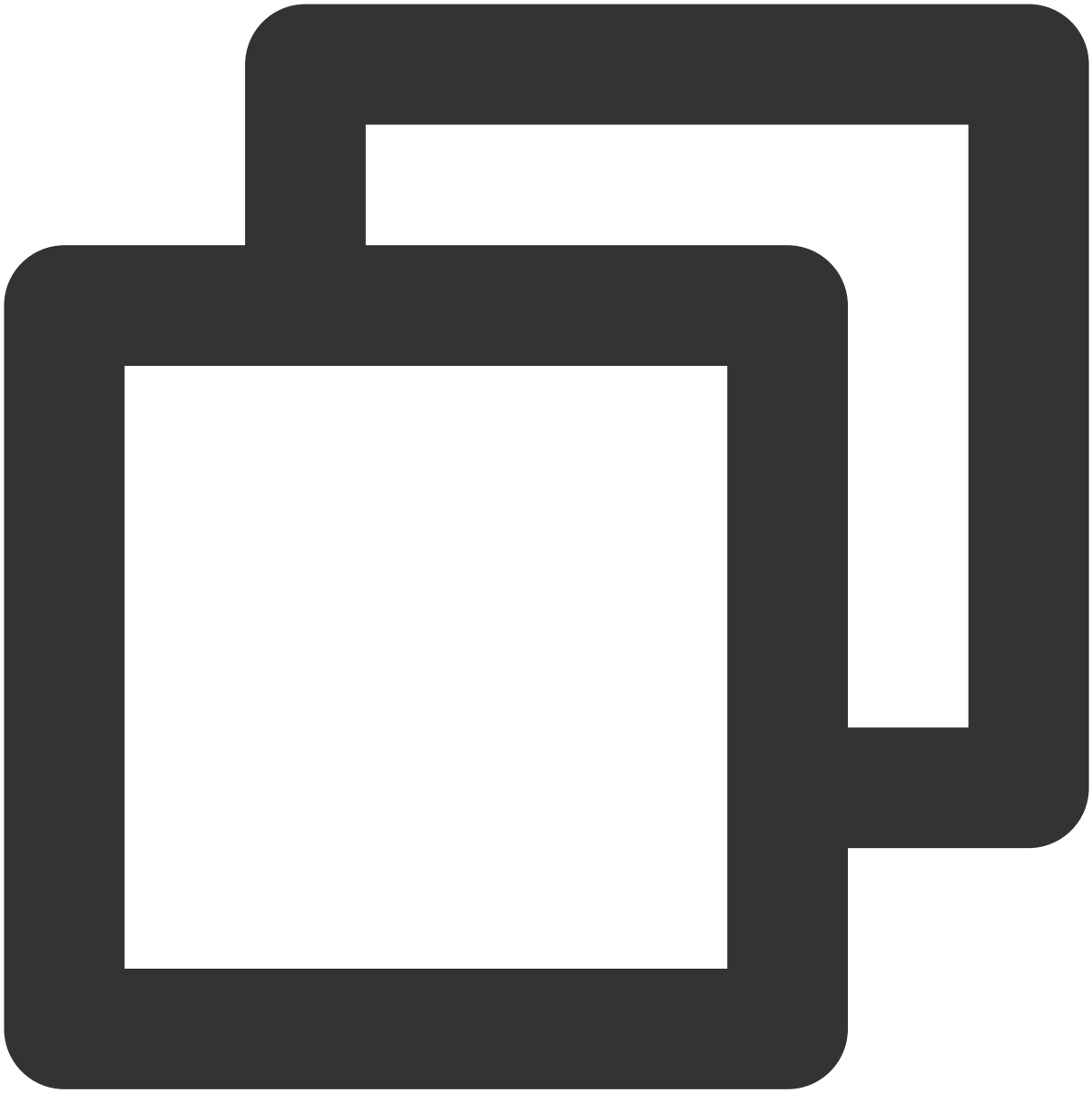
Note:

Private network reporting: Using this reporting method, your service needs to run in the Tencent Cloud VPC. Through VPC connecting directly, you can avoid the security risks of public network communication and save on reporting traffic overhead.

Public network reporting: If your service is deployed locally or in non-Tencent Cloud VPC, you can report data in this method. However, it involves security risks in public network communication and incurs reporting traffic fees.

Connecting Node.js Applications

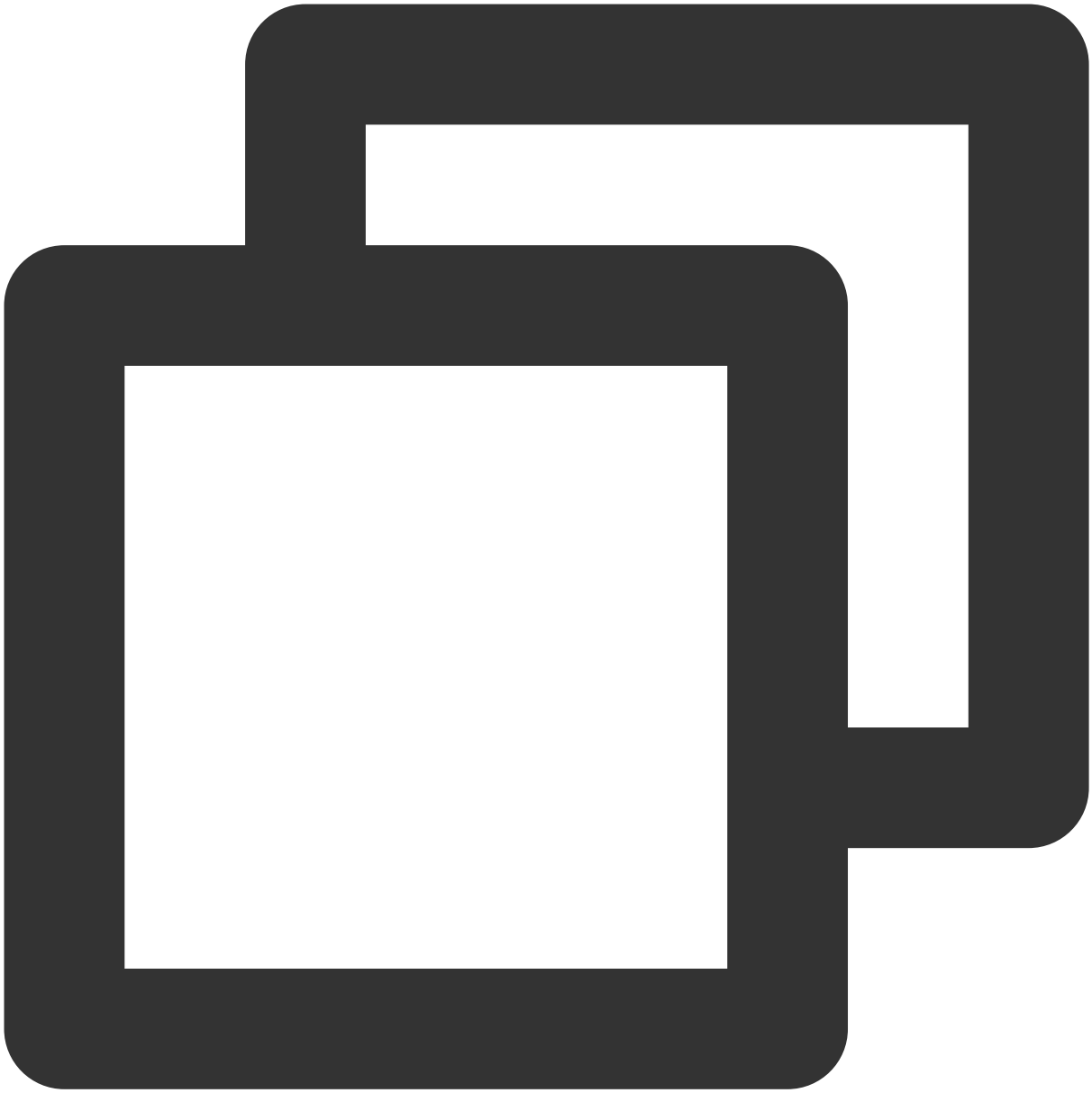
Step 1: Install the required dependency packets.



```
npm install --save @opentelemetry/api
npm install --save @opentelemetry/auto-instrumentations-node
```

Step 2: Add runtime parameters.

Start the Node.js application with the following command:



```
export OTEL_TRACES_EXPORTER="otlp"
export OTEL_RESOURCE_ATTRIBUTES='token=<token>,hostName=<hostName>'
export OTEL_EXPORTER_OTLP_PROTOCOL='grpc'
export OTEL_EXPORTER_OTLP_TRACES_ENDPOINT="<endpoint>"
export OTEL_SERVICE_NAME="<serviceName>"
export NODE_OPTIONS="--require @opentelemetry/auto-instrumentations-node/register"
node main.js
```

The corresponding field descriptions are as follows:

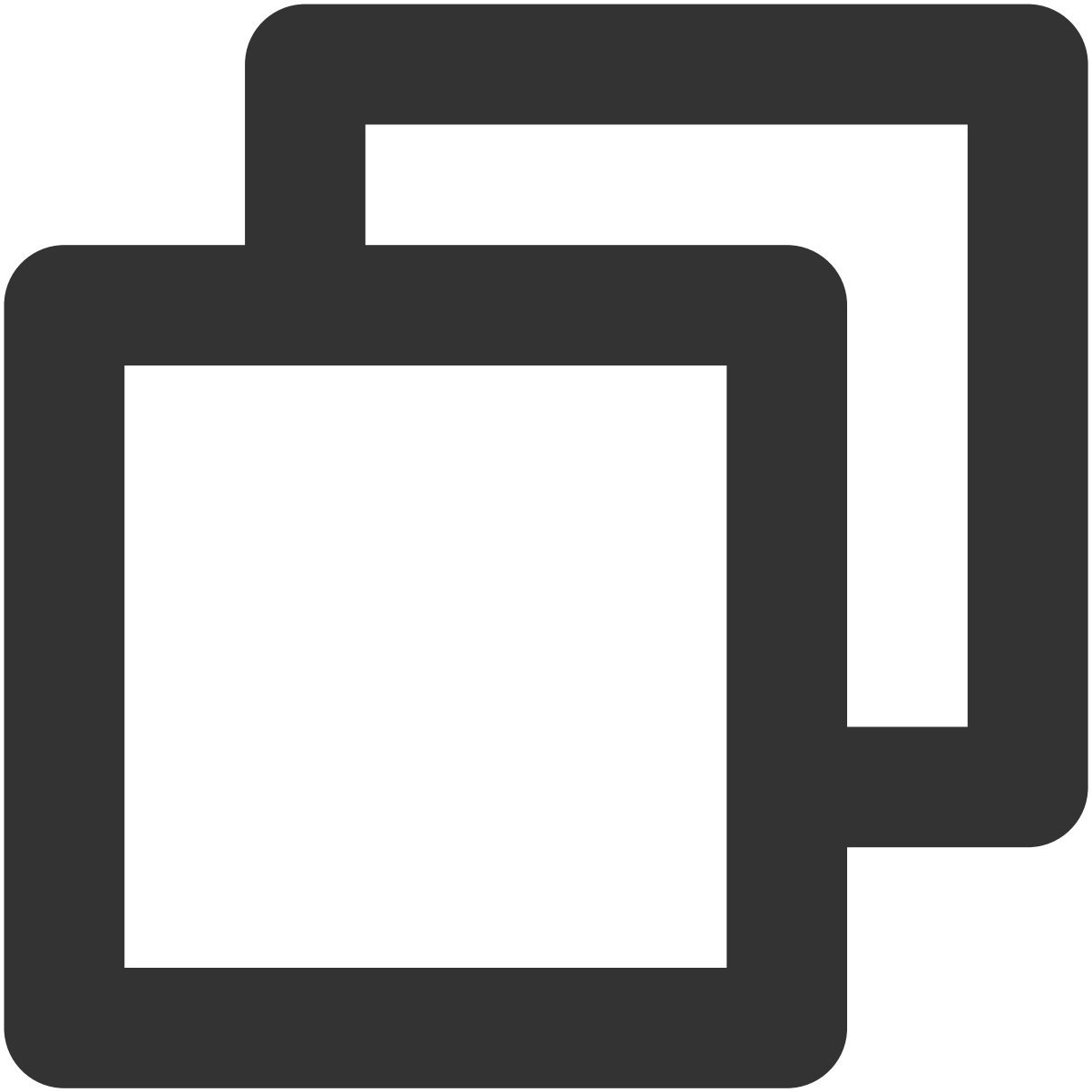
`<serviceName>` : Application name. Multiple application processes connecting with the same serviceName are displayed as multiple instances under the same application in APM. The application name can be up to 63 characters and can only contain lowercase letters, digits, and the separator (-), and it must start with a lowercase letter and end with a digit or lowercase letter.

`<token>` : The business system Token obtained in the preliminary steps.

`<hostName>` : The hostname of this instance, which is the unique identifier of the application instance. It can usually be set to the IP address of the application instance.

`<endpoint>` : The connect point obtained in the preliminary steps.

The following content uses `myService` as the application name, `myToken` as the business system Token, `192.168.0.10` as the hostname, and `http://pl-demo.ap-guangzhou.apm.tencentcs.com:4317` as the example connect point. The complete start-up command is:



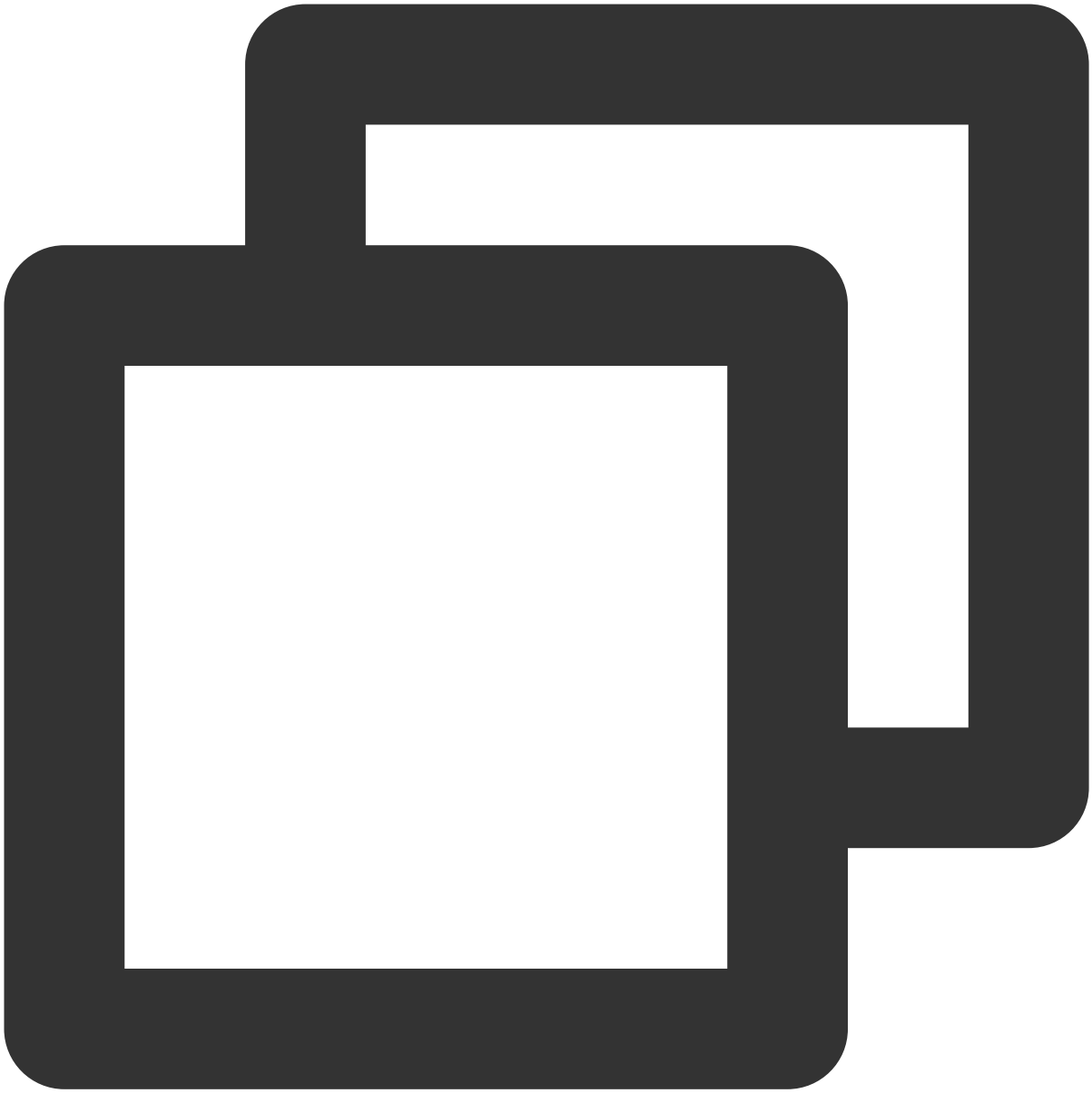
```
export OTEL_TRACES_EXPORTER="otlp"
export OTEL_RESOURCE_ATTRIBUTES='token=myToken,hostName=192.168.0.10'
export OTEL_EXPORTER_OTLP_PROTOCOL='grpc'
export OTEL_EXPORTER_OTLP_TRACES_ENDPOINT="http://pl-demo.ap-guangzhou.apm.tencentc
export OTEL_SERVICE_NAME="myService"
export NODE_OPTIONS="--require @opentelemetry/auto-instrumentations-node/register"
node main.js
```

Connection Verification

After the Node.js application is started, access the corresponding API through port 8080, for example, `https://localhost:8080/` . In the case of normal traffic, APM > Application monitoring > Application List will display the connected applications, and APM > Application monitoring > App details **> Instance monitoring** will display the connected application instances. Since there is some latency in processing observable data, if the application or instance is not found in the console after connecting, wait about 30 seconds.

Custom Event Tracking (Optional)

When automatic instrumentation does not meet your scenarios, or you need to add business layer instrumentation, you can see the following content and use the OpenTelemetry API to add custom instrumentation. This document only shows the most basic custom Event Tracking method. The OpenTelemetry community offers more flexible custom instrumentation methods, and you can see the OpenTelemetry community-provided Javascript Custom Event Tracking Documentation for specific methods.



```
const opentelemetry = require("@opentelemetry/api")

app.get("/attr", async(req, res) => {
  const tracer = opentelemetry.trace.getTracer(
    'my-service-tracer'
  );
  tracer.startActiveSpan('new internal span', span => {
    span.addEvent("Acquiring lock", {
      'log.severity':'error',
      'log.message':'data node found',
    })
```

```
    span.addEvent("Got lock, doing work...", {
      'log.severity':'11111',
      'log.message':'2222222',
      'log.message1':'3333333',
    })
    span.addEvent("Unlocking")
    span.end();
  });
  res.json({ code: 200, msg: "success" });
})
```

Reporting with Native Jaeger SDK

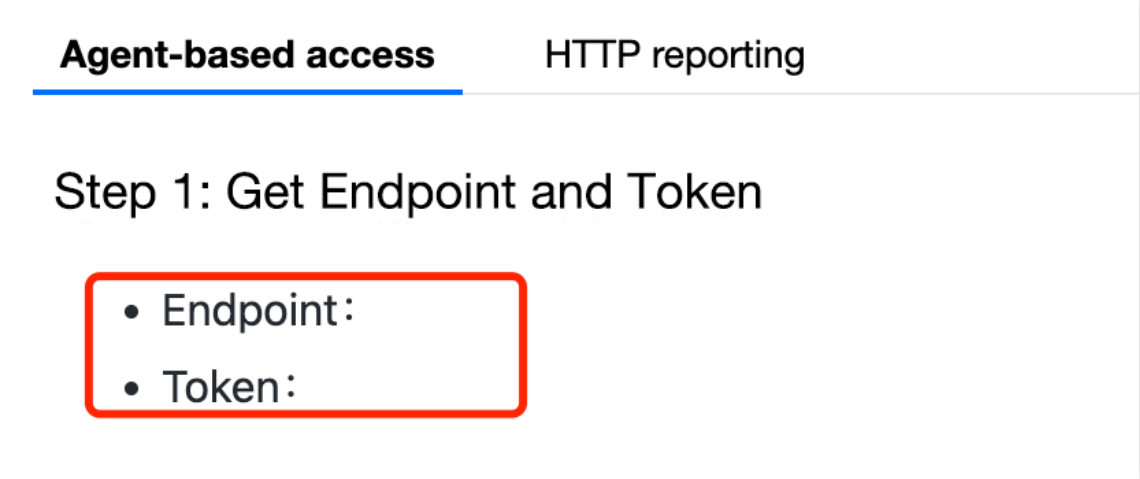
Last updated：2023-12-25 16:03:23

This document describes how to report the data of a Node.js application with the native Jaeger SDK.

Directions

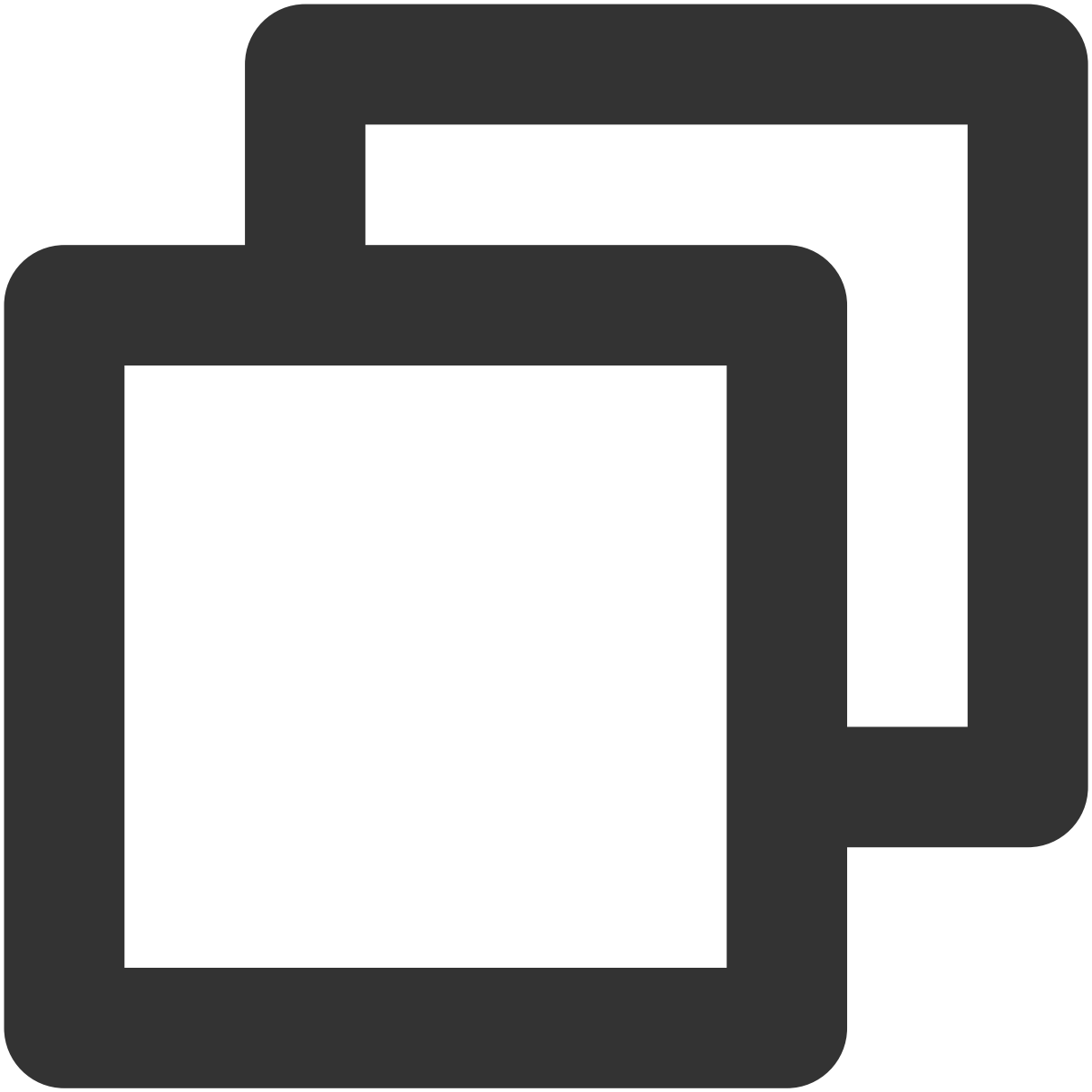
Step 1. Get the endpoint and token

Log in to the APM console, enter the **Application monitoring** > **Application list** page, click **Access application**, and select the Node.js language and Jaeger data collection method. Then, get the endpoint and token in the step of access method selection.



Step 2. Install dependencies

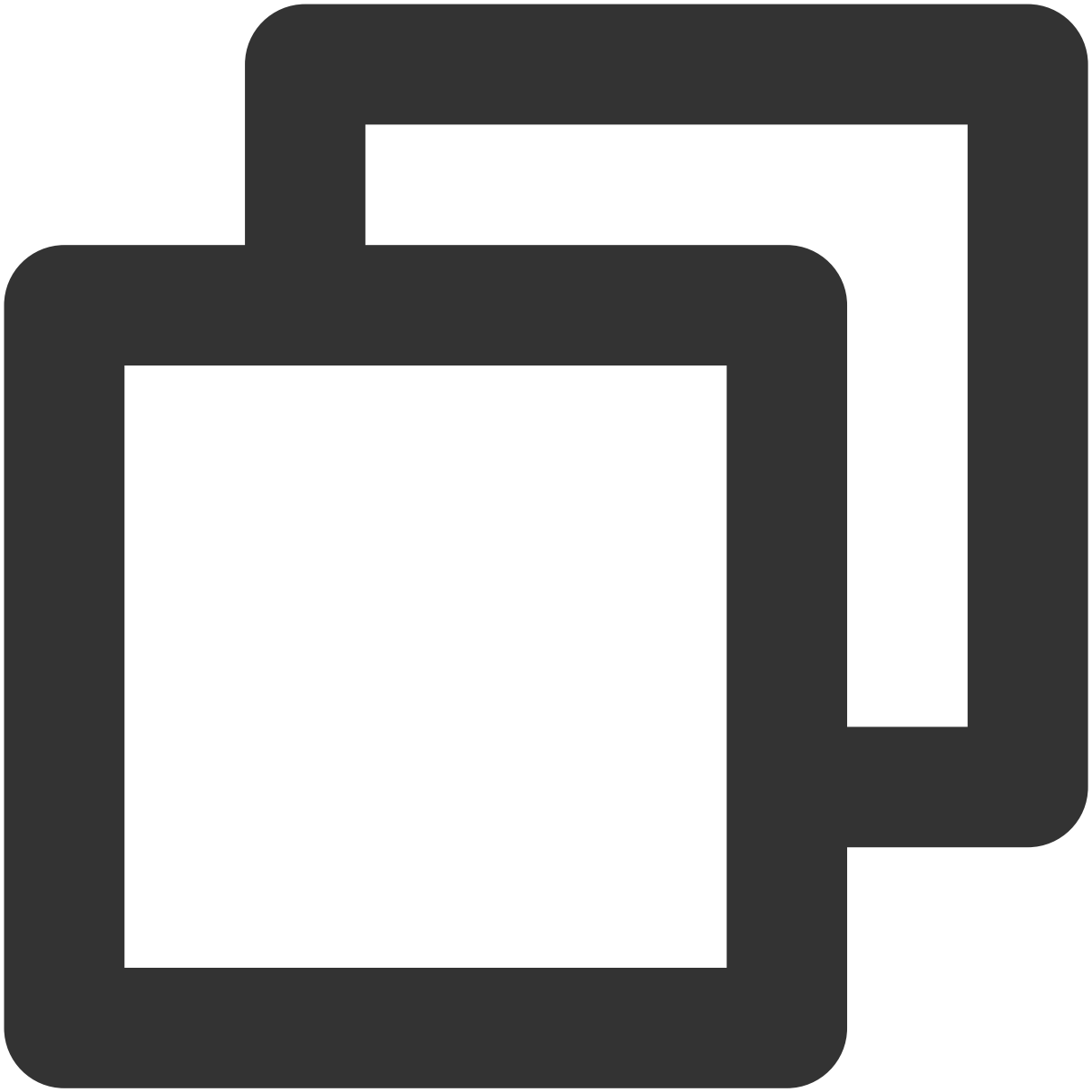
Install dependencies by using npm.



```
$ npm i jaeger-client
```

Step 3. Import the SDK and report data

1. Import the SDK. Below is a sample:



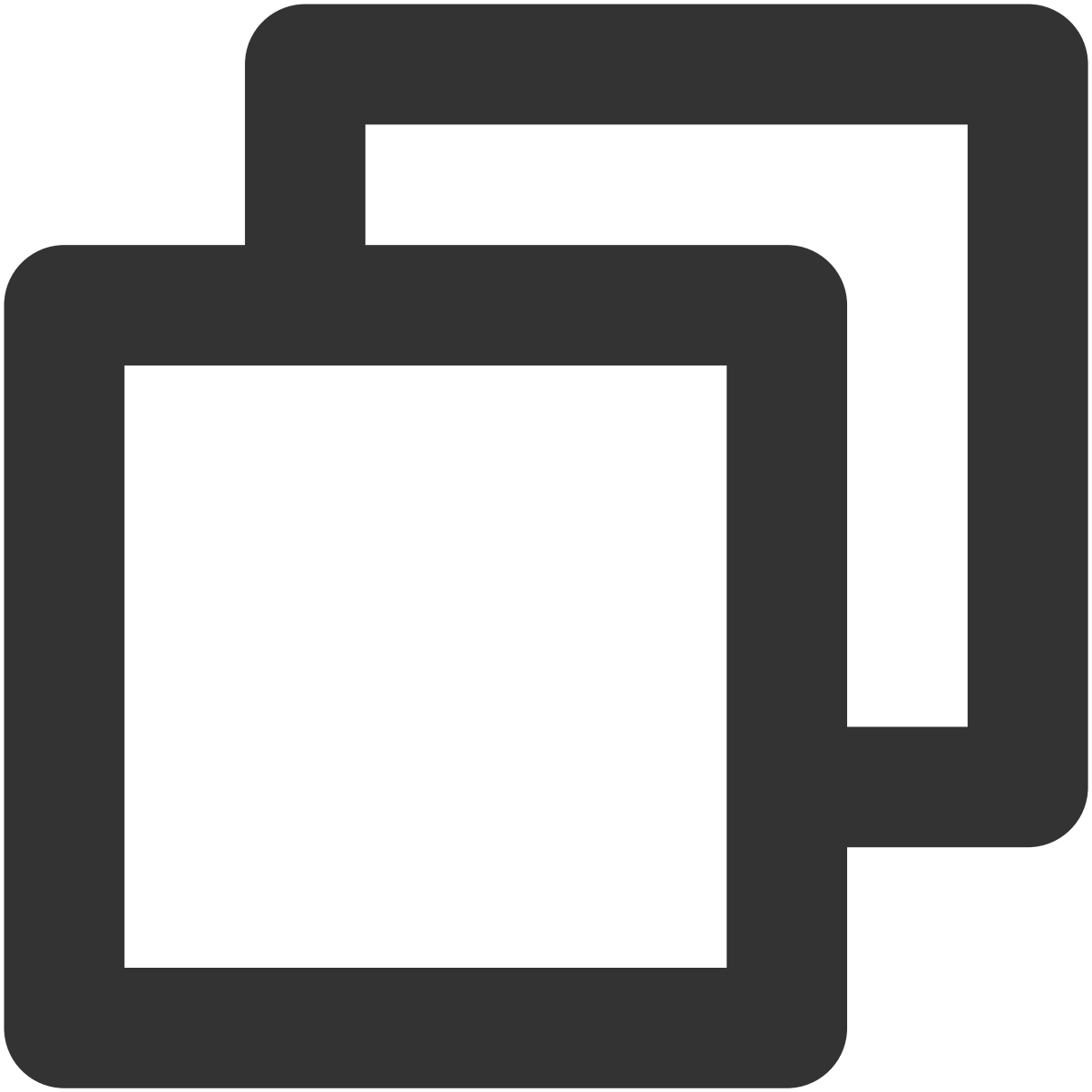
```
const initTracer = require('jaeger-client').initTracer;

// Jaeger configuration
const config = {
 serviceName: 'service-name', // Customizable service name
 sampler: {
     type: 'const',
     param: 1,
 },
 reporter: {
     logSpans: true,
```

```
      collectorEndpoint: 'http://ap-guangzhou.apm.tencentcs.com:14268/api/traces', /
  },
};

const options = {
 tags: {
      token: 'Vds************CrKck' // The requested token
  },
};
```

Note:

Node.js uses API to directly report data, so there is no need to start the Jaeger agent. Select the endpoint of your network environment and add the suffix `/api/traces` to form the actual endpoint.

2. Report data. Below is a sample:



```
// Initialize the tracer instance object
const tracer = initTracer(config, options);

// Initialize the span instance object
const span = tracer.startSpan('spanStart');

// Mark the current service as the server
span.setTag('span.kind', 'server');

// Set one or more tags (optional)
span.setTag('tagName', 'tagValue');
```

```
// Set one or more events (optional)
span.log({ event: 'timestamp', value: Date.now() });

// Mark the end of the span
span.finish();
```

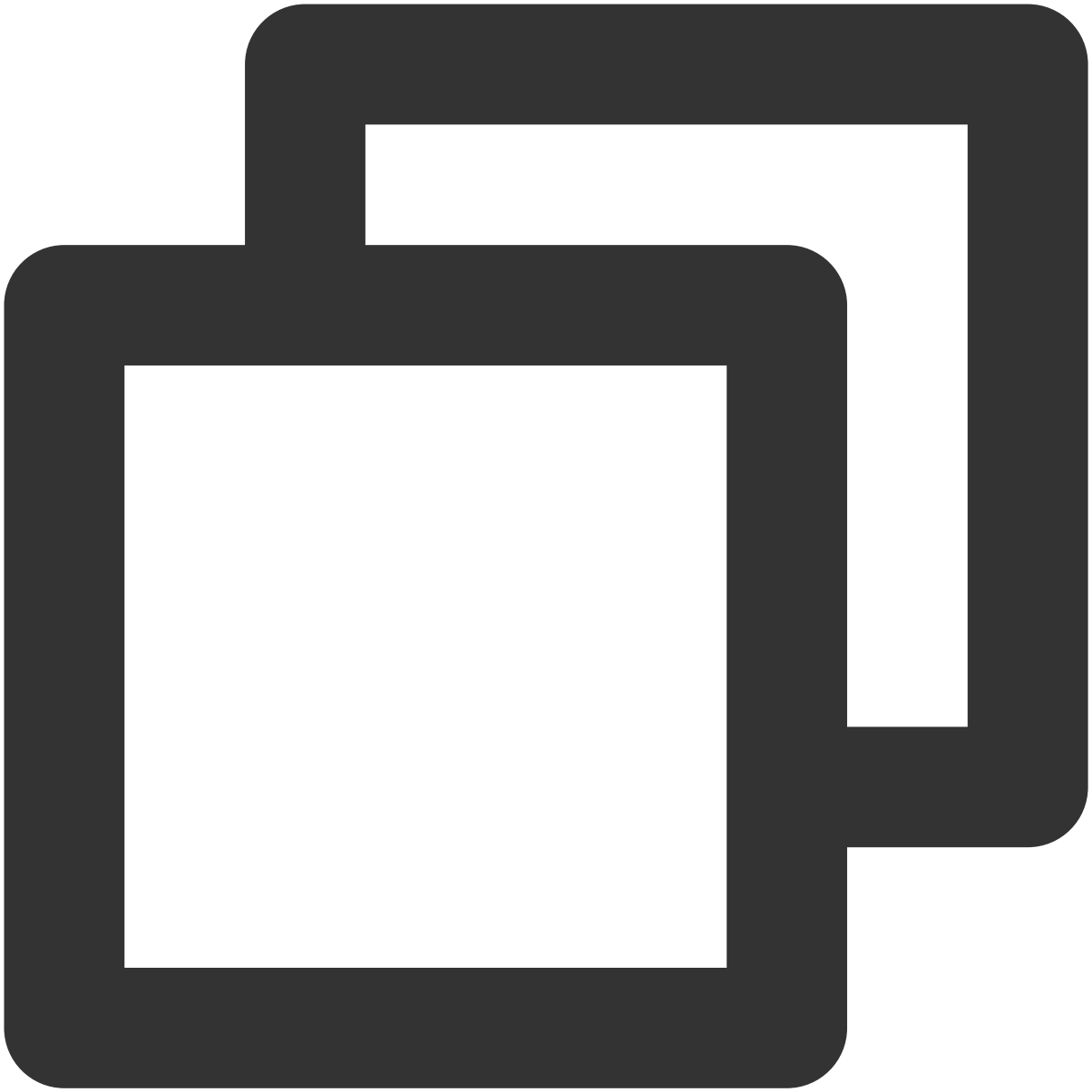
Accessing PHP Application
Connecting PHP Application via OpenTelemetry-PHP (Recommended)

Last updated：2024-07-08 11:27:23

Note:
OpenTelemetry is a collection of tools, APIs, and SDKs for monitoring, generating, collecting, and exporting telemetry data (metrics, logs, and traces) to help users analyze the performance and behavior of the software. For more information about OpenTelemetry, see the OpenTelemetry official website.
The OpenTelemetry community is active, with rapid technological changes, and widely compatible with mainstream programming languages, components, and frameworks, making its link-tracing capability highly popular for cloud-native microservices and container architectures.
This document introduces how to integrate PHP applications using the community's OpenTelemetry-PHP scheme through relevant operations.
The OpenTelemetry-PHP scheme provides automatic event tracking for commonly used PHP dependency libraries and frameworks, such as Slim, without modifying the code to report linkage information. For other dependency libraries and frameworks that support automatic event tracking, see complete list provided by the OpenTelemetry community.

Prerequisites

Install the following tools:
PECL
composer
Ensure that you can run the following commands in the shell:



```
php -v
composer -v
```

Automatic Integration

PHP 8.0+

For details on the list of frameworks supported by automatic event tracking, see OpenTelemetry official documentation.

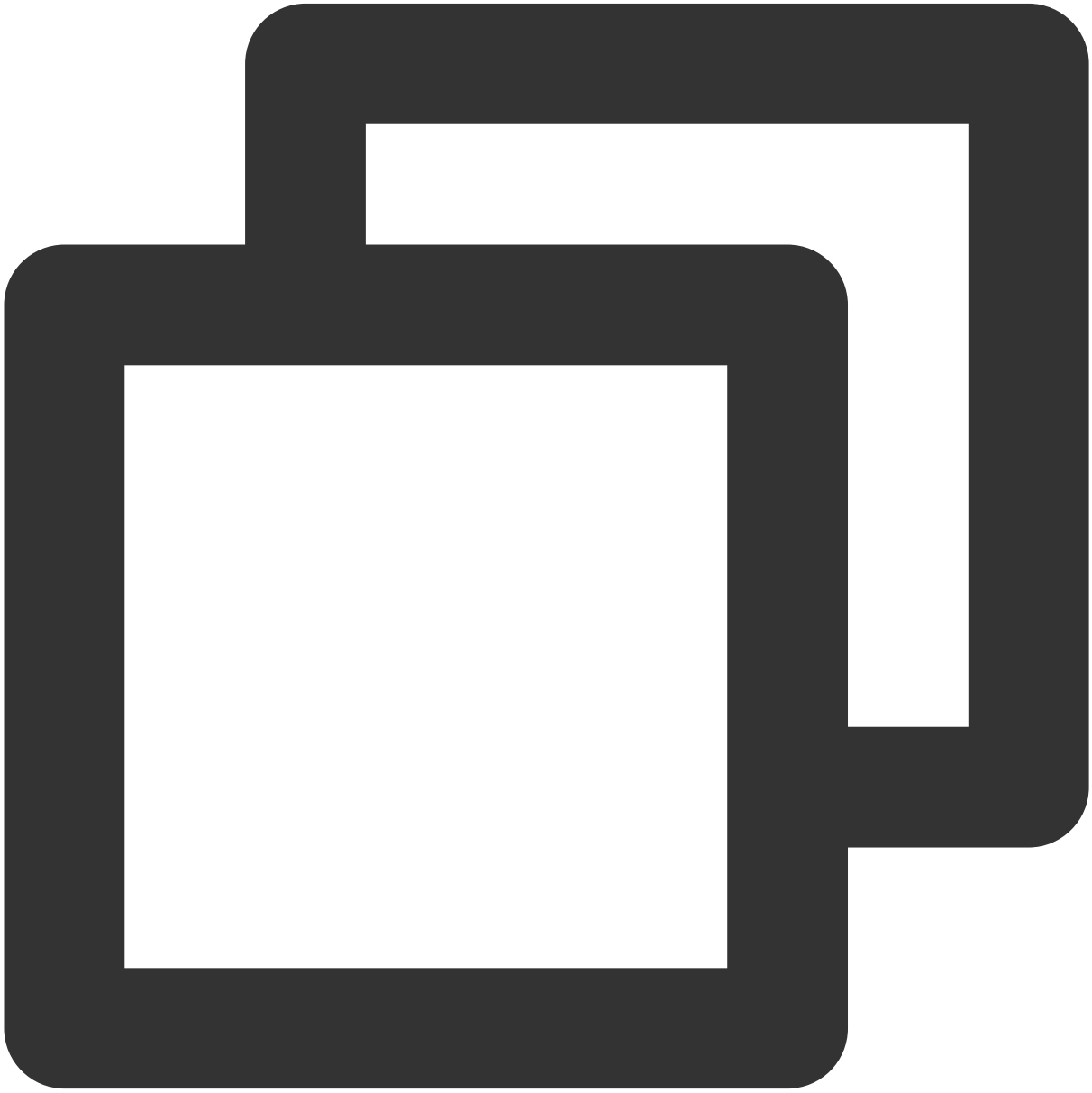
Manual Integration

PHP 7.4+

Demo Application

The sample code `index.php` is an HTTP Server using PDO to connect to a MySQL database for database operations. Set up the corresponding MySQL service yourself or purchase cloud services directly.

1. Initializing Application



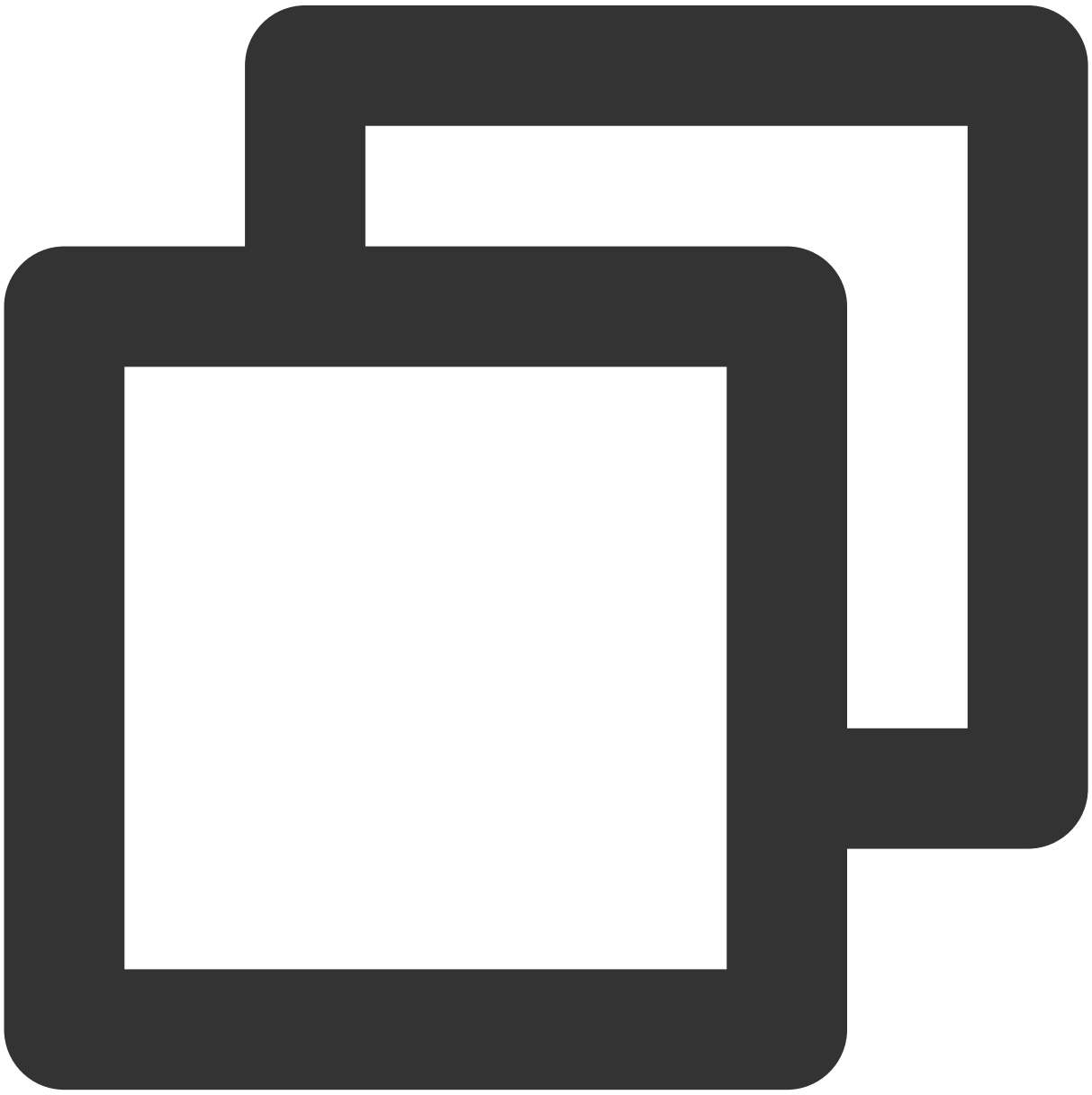
```
mkdir <project-name> && cd <project-name>
```

```
composer init \\
 --no-interaction \\
 --stability beta \\
 --require slim/slim:"^4" \\
 --require slim/psr7:"^1"
composer update
```

2. Writing Business Code

Create an `index.php` file in the <project-name> directory and add the following content.

The following content will simulate a search operation using PDO to connect to MySQL with an HTTP Server API.



```
<?php
use Psr\\Http\\Message\\ResponseInterface as Response;
use Psr\\Http\\Message\\ServerRequestInterface as Request;
use Slim\\Factory\\AppFactory;

require __DIR__ . '/vendor/autoload.php';

$app = AppFactory::create();

$app->get('/getID', function (Request $request, Response $response) {

    $dbms = 'mysql';          // Database type
    $host = 'localhost';      // Database hostname
    $dbName = 'Mydb';         // Database in use
    $user = 'root';           // Database connection username
    $pass = '';               // Corresponding password
    $dsn = "$dbms:host=$host;dbname=$dbName";

    try {
        $dbh = new PDO($dsn, $user, $pass); // Initialize a PDO object
        echo "Connection successful<br/>";

        foreach ($dbh->query('SELECT id from userInfo') as $row) {
            $response->getBody()->write($row[0] . "<br/>");
        }

        $dbh = null;
    } catch (PDOException $e) {
        die ("Error!: " . $e->getMessage() . "<br/>");
    }

    return $response;
});

$app->run();
```

Preliminary steps: Get the access point and token.

1. Log in to the TCOP console.
2. In the left sidebar, select **Application Performance Management > Application monitoring**, and click **Application list > Access application**.
3. In the **Data Ingestion** drawer that pops up on the right, click **PHP** language.
4. On the **Integrate PHP Applications** page, select the desired **Region** and **Business System**.

5. Select **Access protocol type** as **OpenTelemetry**.
6. **Reporting method** Choose your desired reporting method, and obtain your **Access Point** and **Token**.

Note:

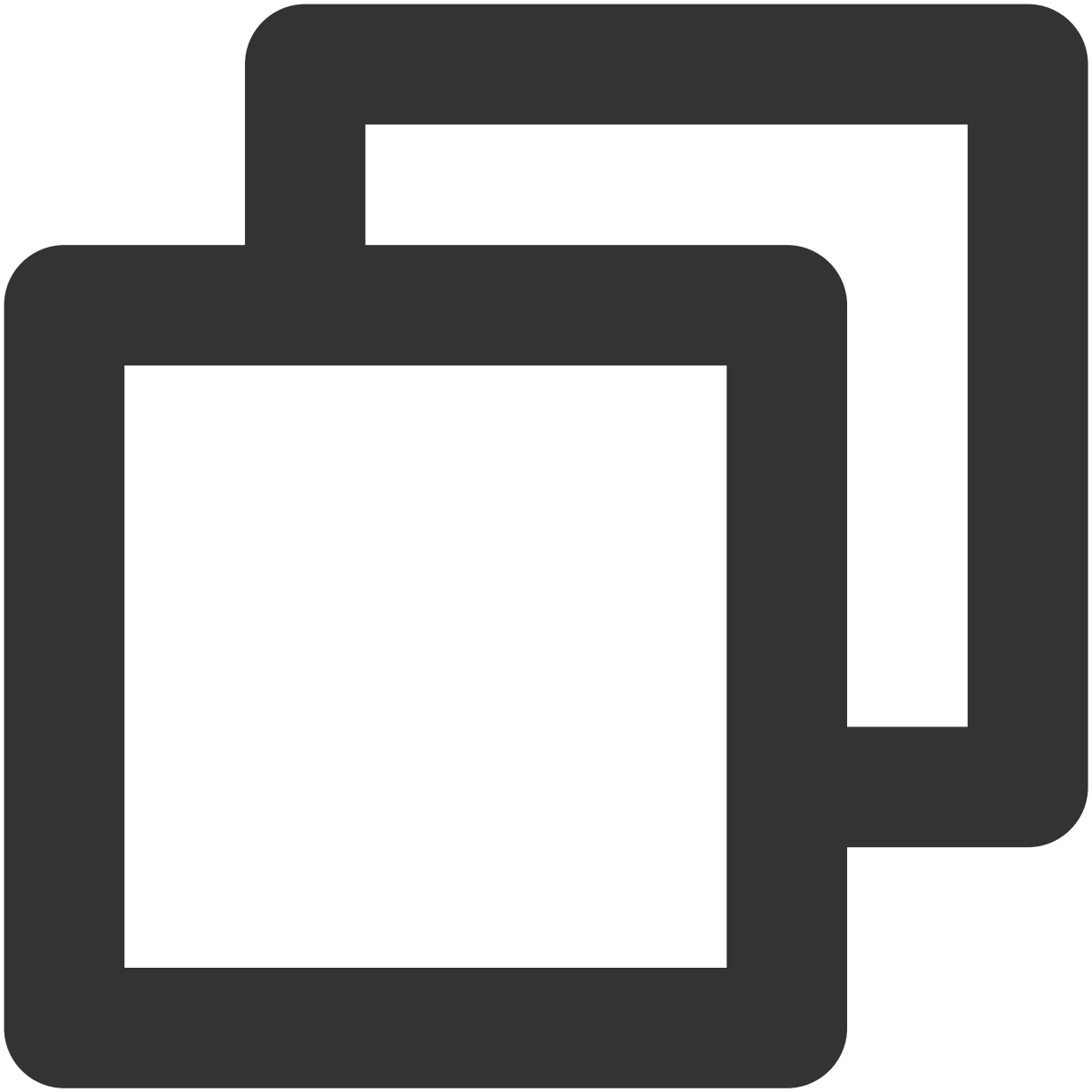
Private network reporting: Using this reporting method, your service needs to run in the Tencent Cloud VPC. Through VPC connecting directly, you can avoid the security risks of public network communication and save on reporting traffic overhead.
Public network reporting: If your service is deployed locally or in non-Tencent Cloud VPC, you can report data in this method. However, it involves security risks in public network communication and incurs reporting traffic fees.

Automatic Integration Scheme (Recommended)

Step 1: Build OpenTelemetry PHP extension.

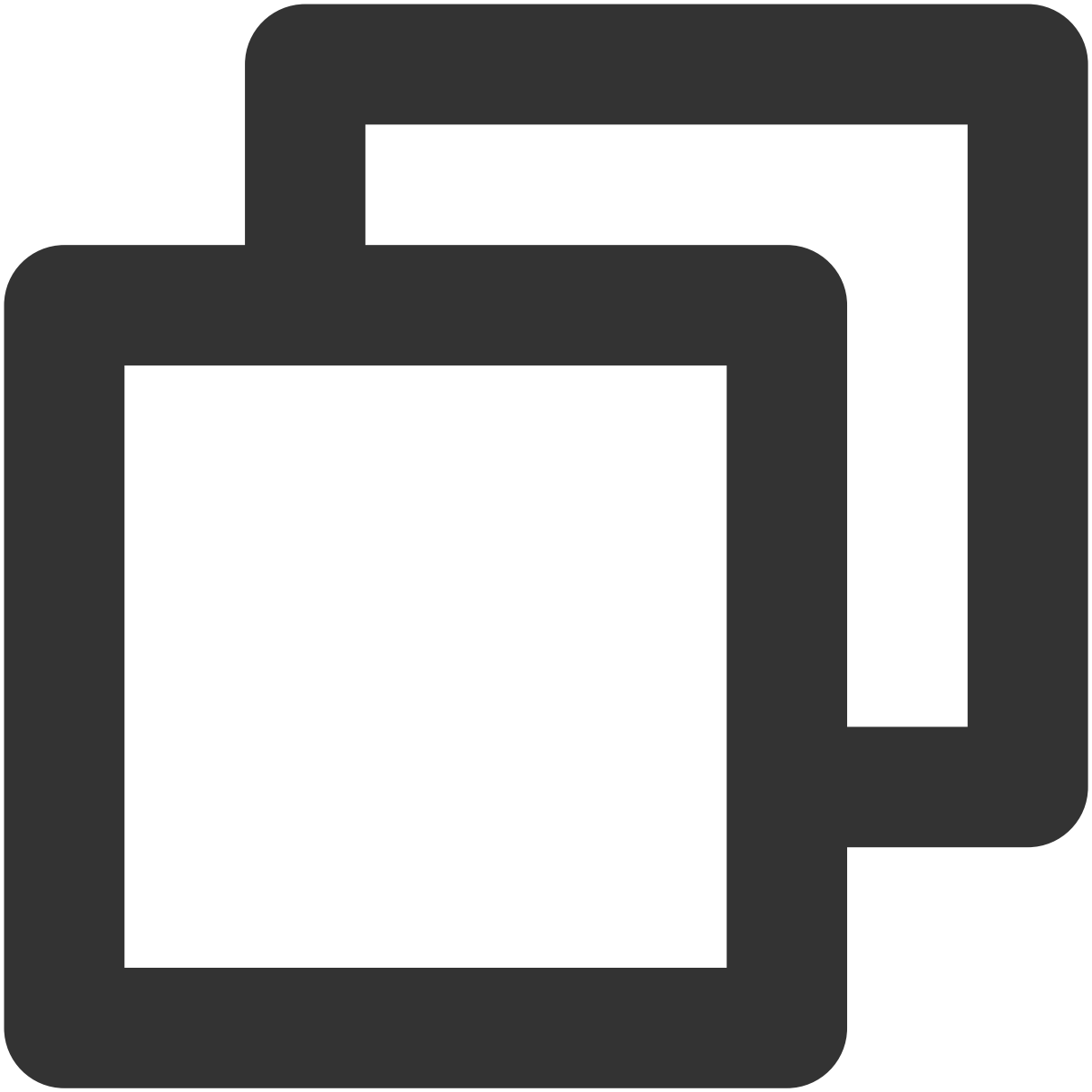
Note:

If you have already built the OpenTelemetry PHP extension, you can skip this step.
1. Download the tools required to build the OpenTelemetry PHP extension:
macOS



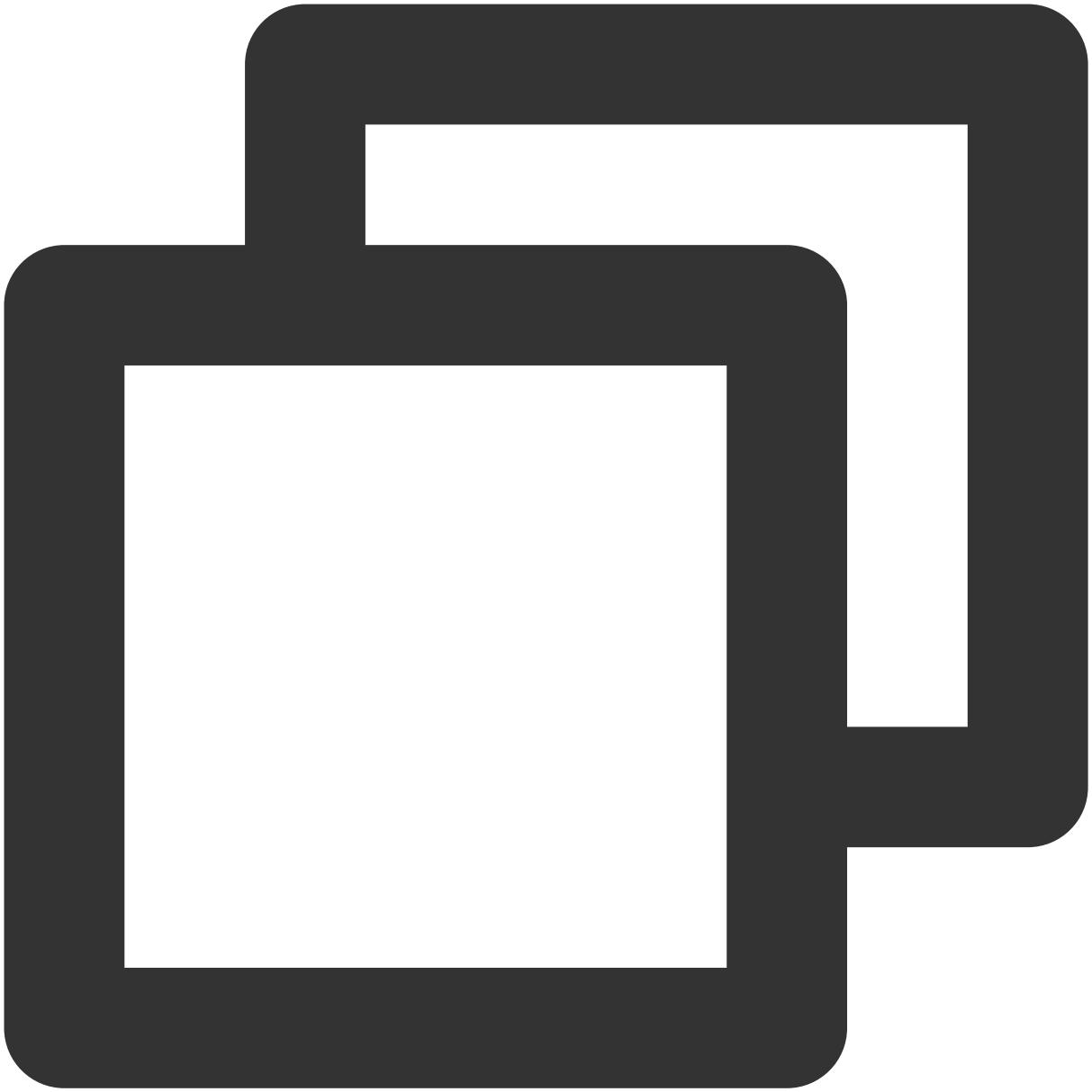
```
brew install gcc make autoconf
```

Linux (apt)



```
sudo apt-get install gcc make autoconf
```

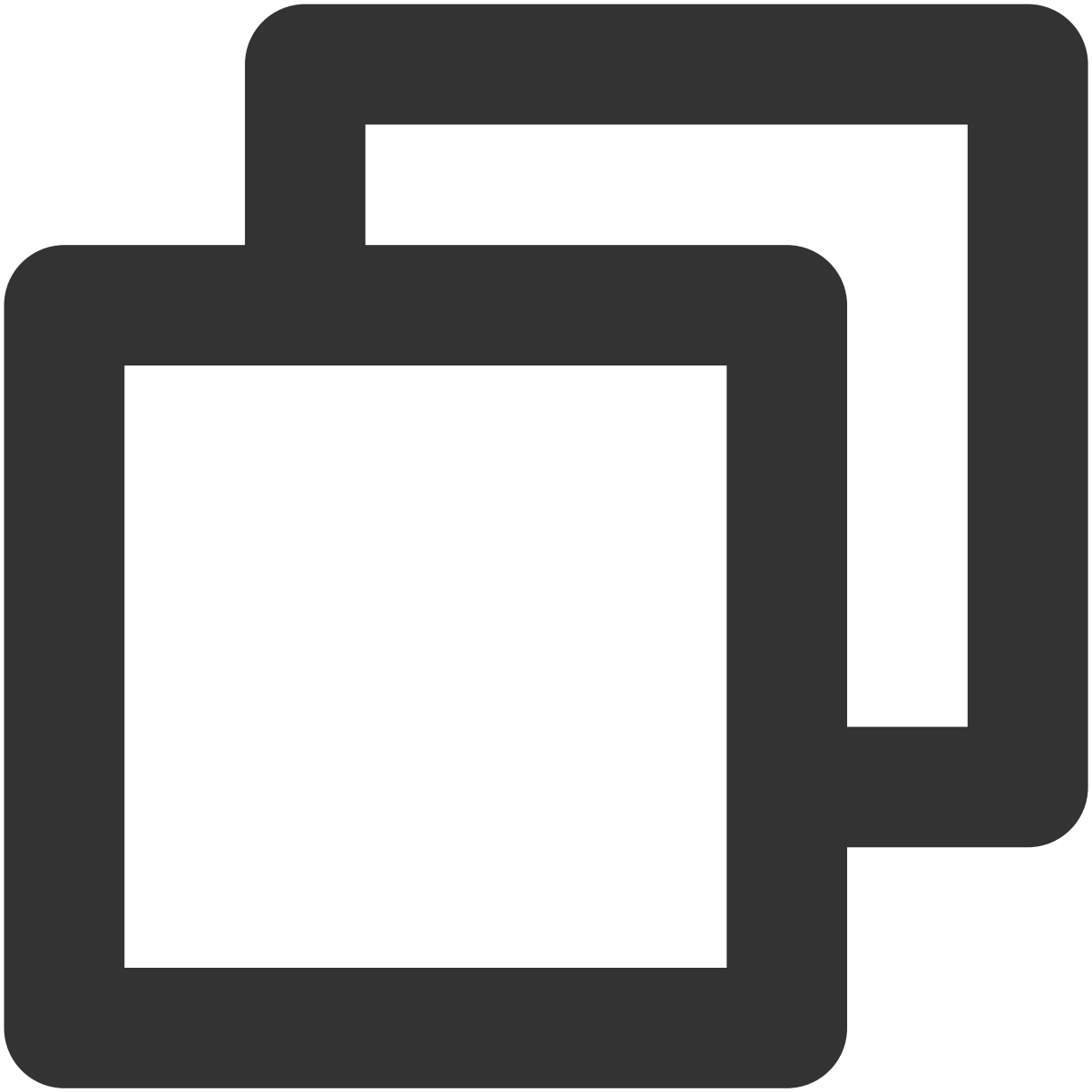
2. Build the OpenTelemetry PHP extension using PECL:



```
pecl install opentelemetry
```

Note:

The last few lines of output upon successful build are as follows (paths may vary):



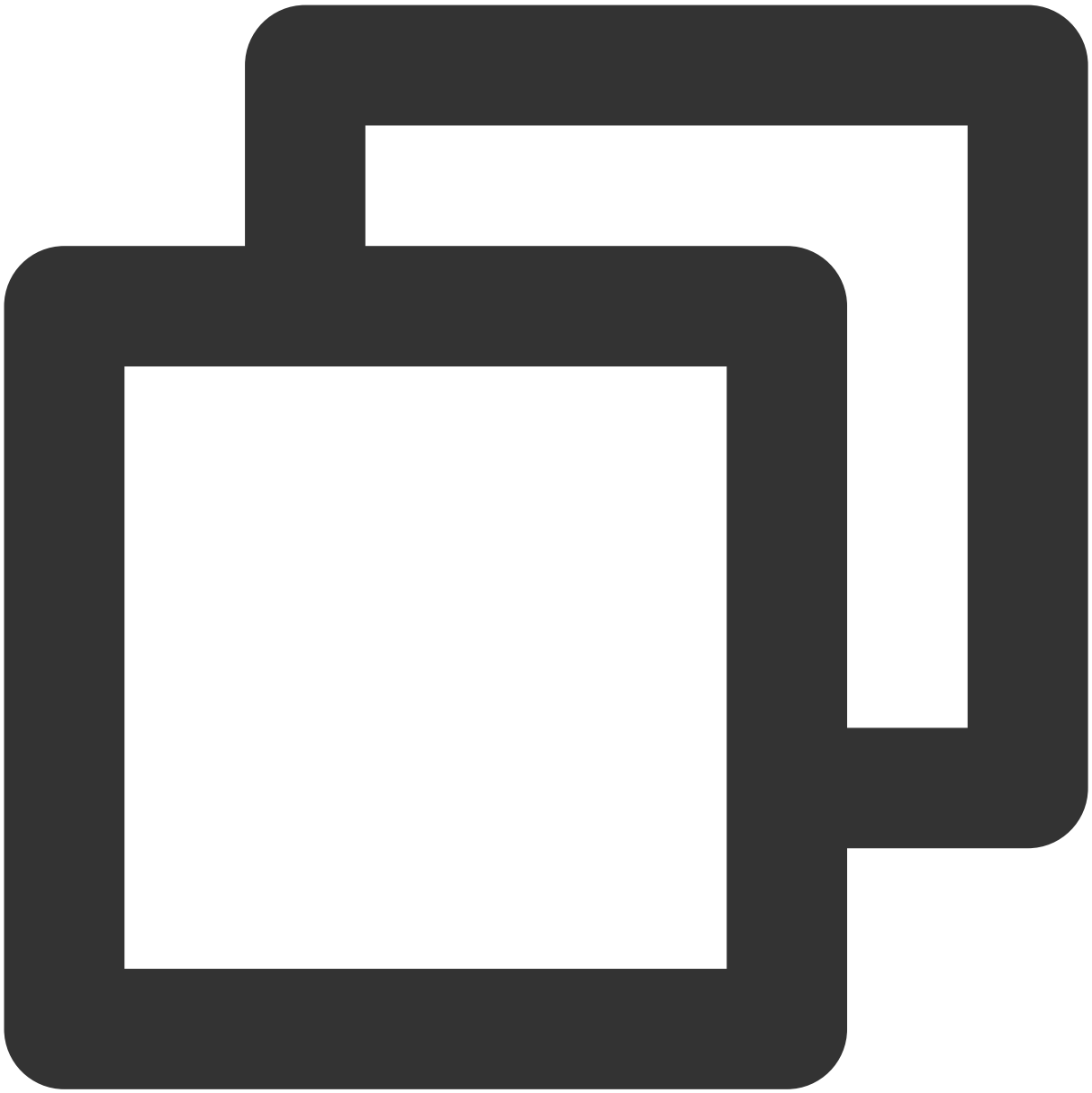
```
Build process completed successfully
Installing '/opt/homebrew/Cellar/php/8.2.8/pecl/2020829/opentelemetry.so'
install ok: channel://pecl.php.net/opentelemetry-1.0.3
Extension opentelemetry enabled in php.ini
```

3. Enable the OpenTelemetry PHP extension.

Note:

If `Extension opentelemetry enabled in php.ini` is output in the previous step, it is enabled. Skip this step.

Add the following contents to the `php.ini` file:



```
[opentelemetry]
extension=opentelemetry.so
```

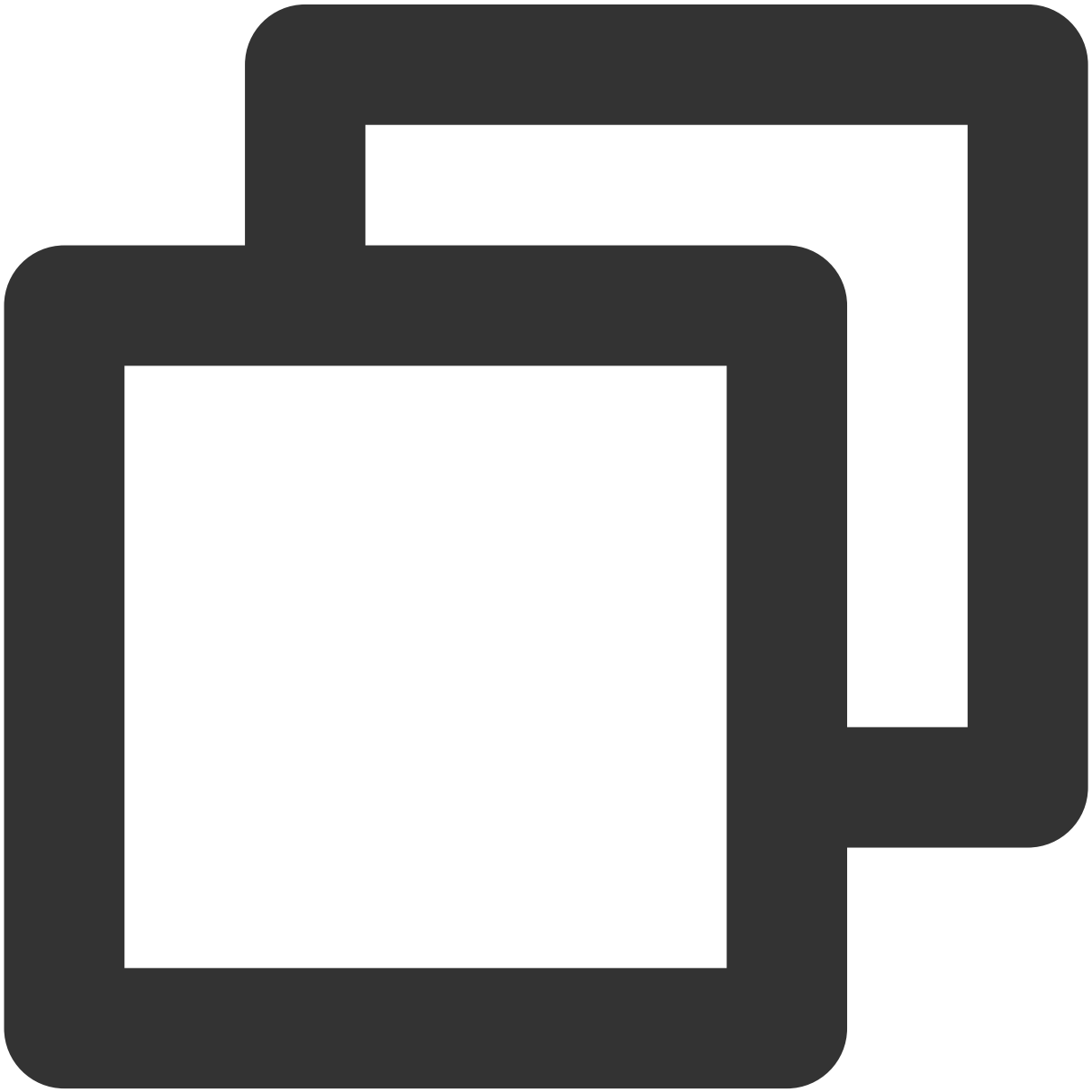
`php.ini` file locations may include:

OS	PATH
Linux	/etc/php.ini /usr/bin/php5/bin/php.ini

	/etc/php/php.ini /etc/php5/apache2/php.ini
Mac OSX	/private/etc/php.ini
Windows (with XAMPP installed)	C:/xampp/php/php.ini

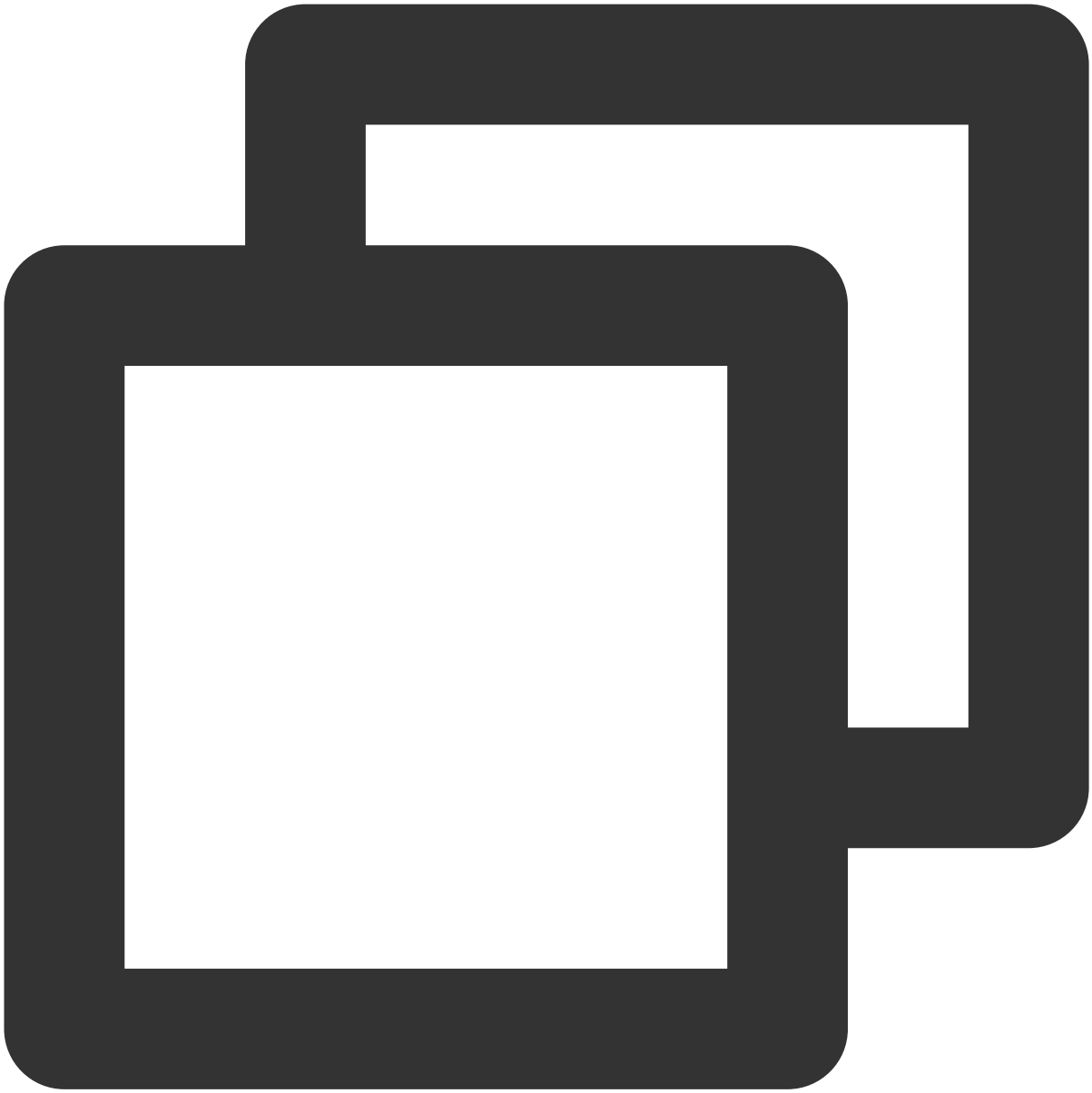
4. Verify that the build and enablement were successful.

Solution 1:



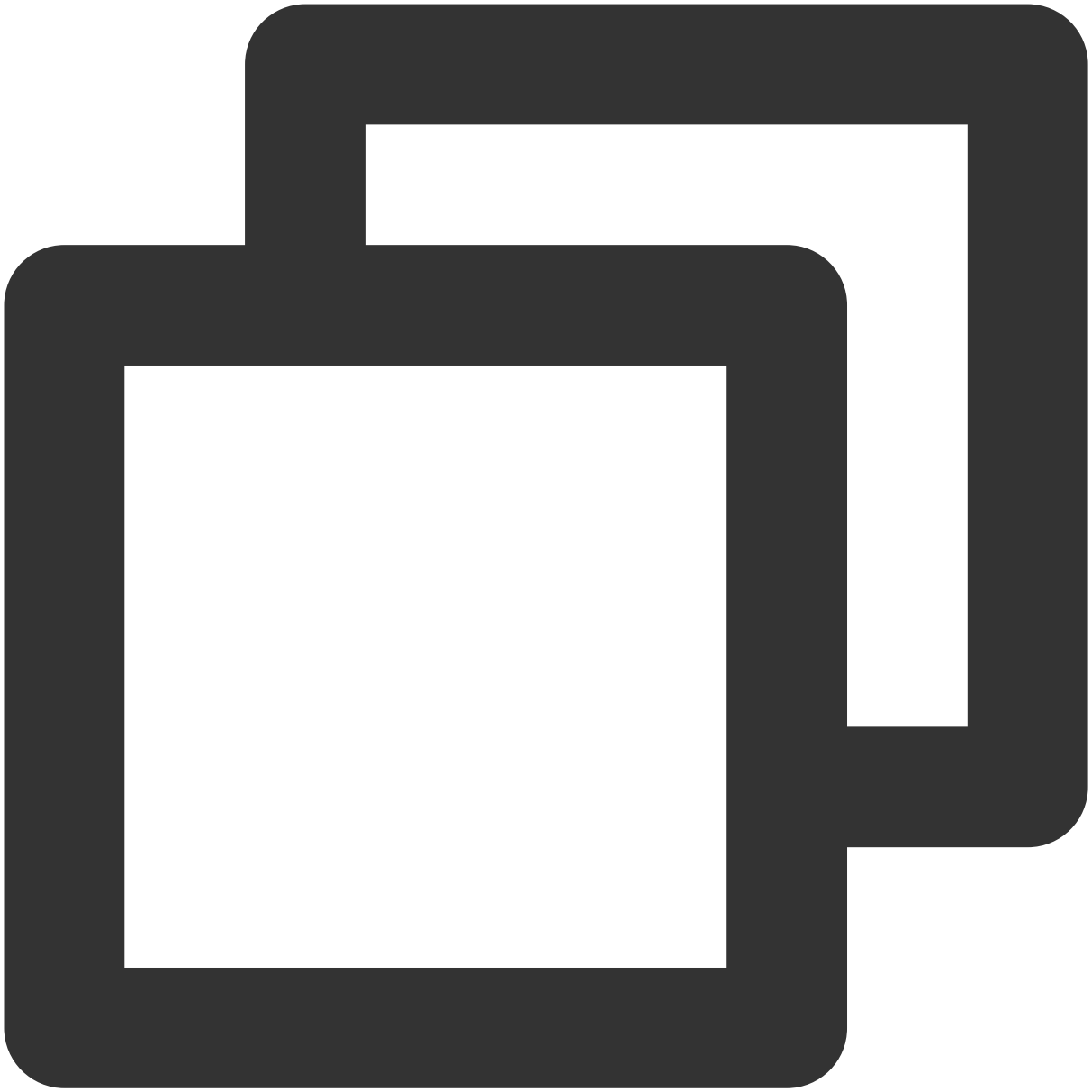
```
php -m | grep opentelemetry
```

Expected output:



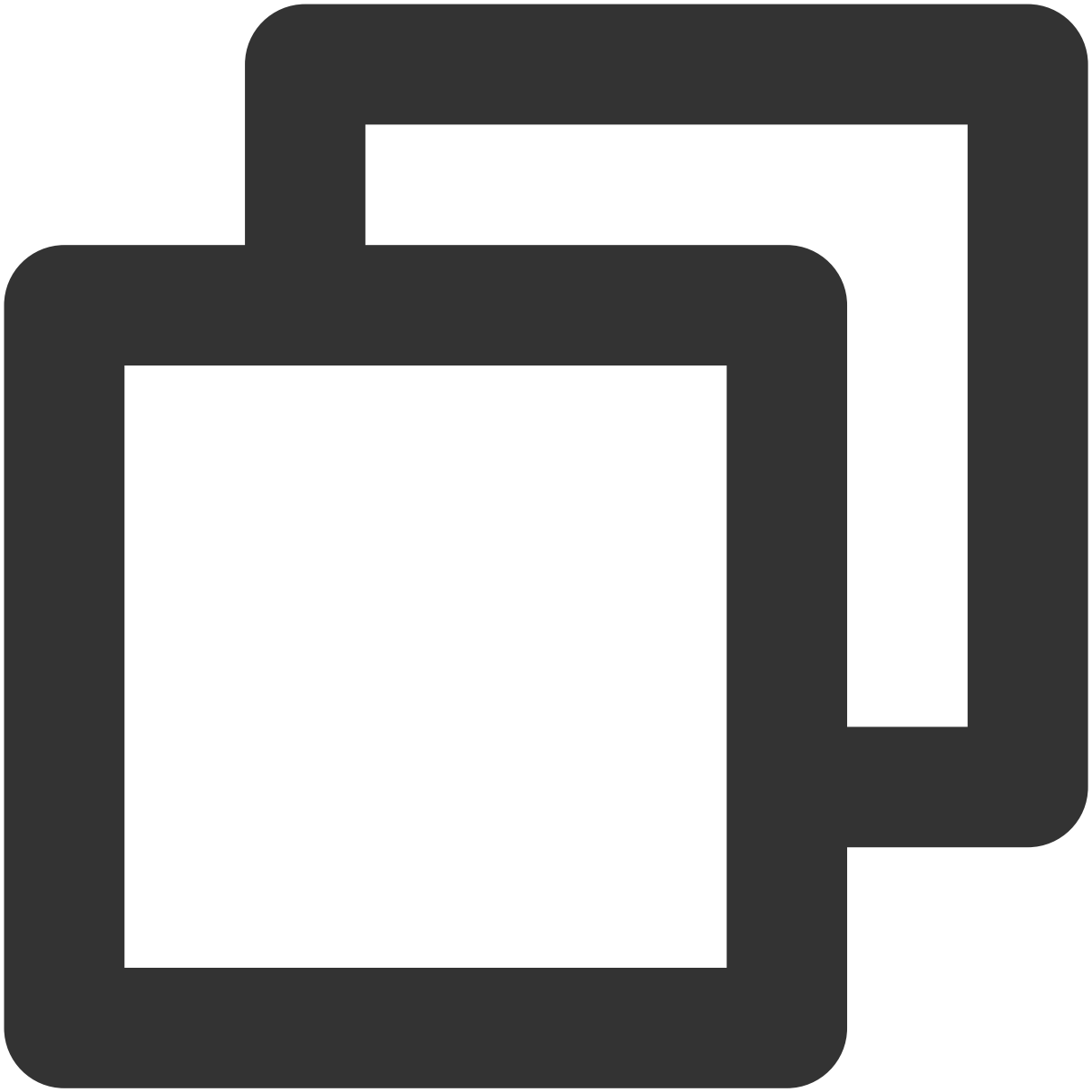
```
opentelemetry
```

Solution 2:



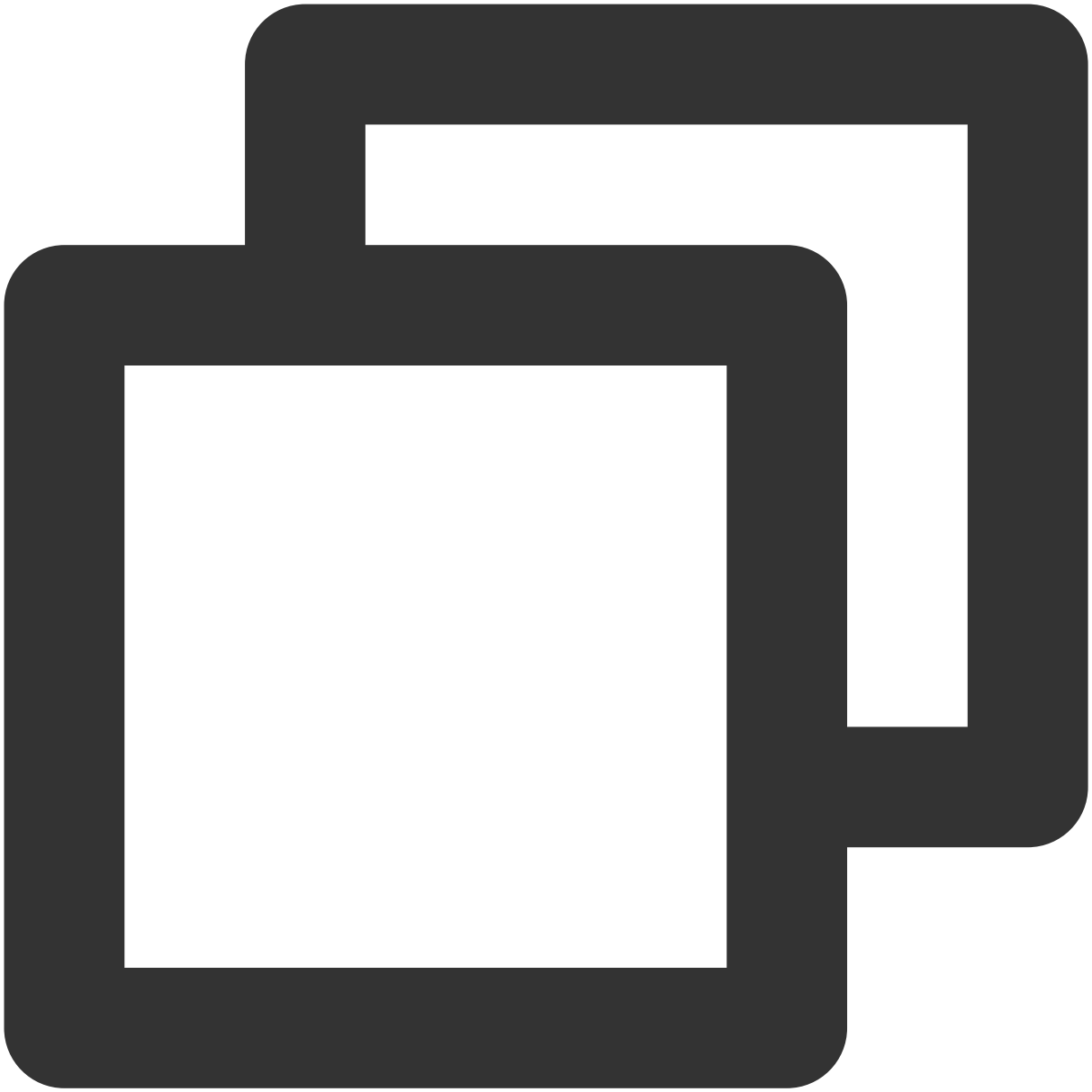
```
php --ri opentelemetry
```

Expected output:



```
opentelemetry
opentelemetry support => enabled
extension version => 1.0.3
```

5. Add additional dependencies required for OpenTelemetry PHP automatic event tracking to the application.



```
pecl install grpc # This step takes a long time to build.

composer require \\
  open-telemetry/sdk \\
  open-telemetry/exporter-otlp \\
  open-telemetry/transport-grpc \\
  php-http/guzzle7-adapter \\
  open-telemetry/opentelemetry-auto-slim \\
  open-telemetry/opentelemetry-auto-pdo
```

open-telemetry/sdk: OpenTelemetry PHP SDK.

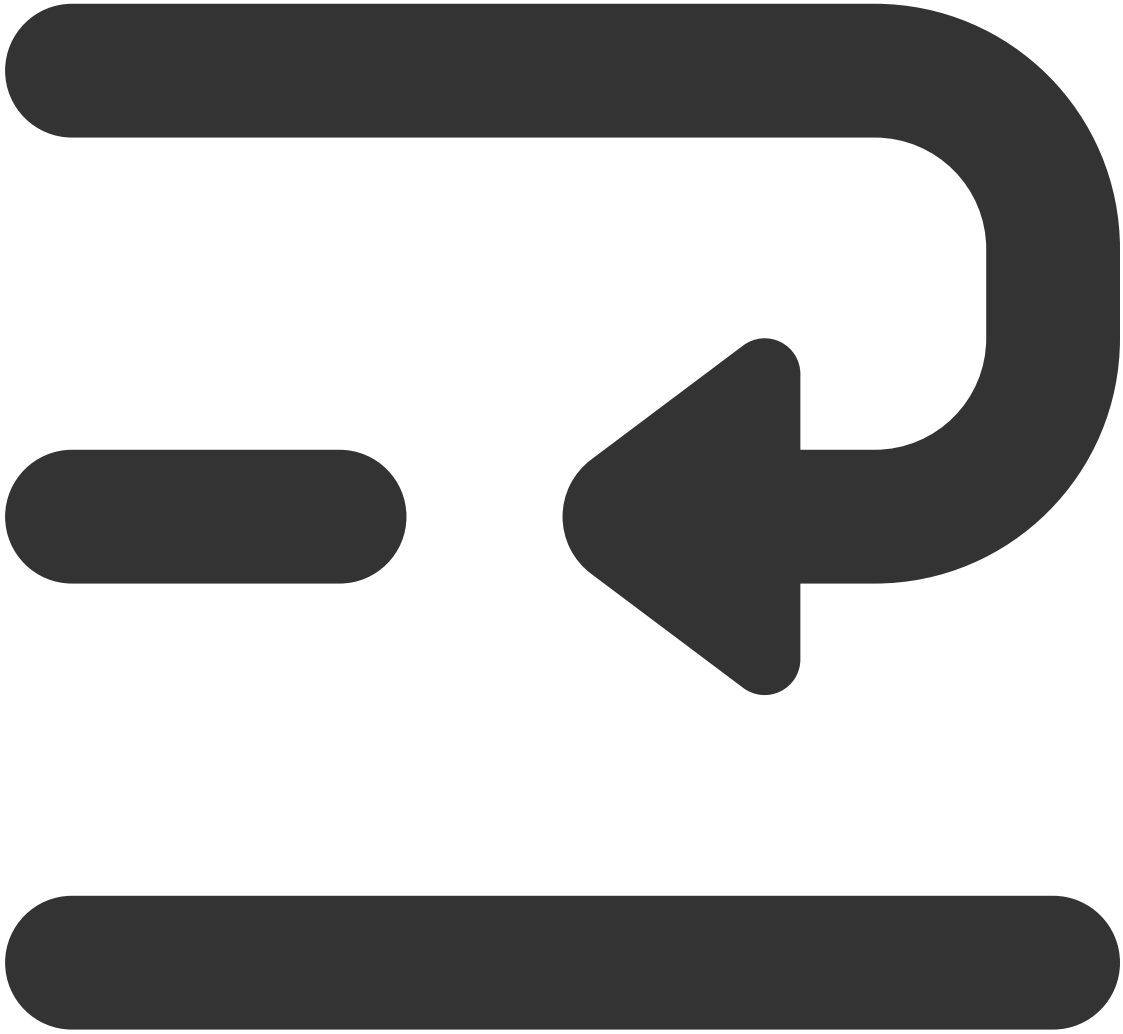
open-telemetry/exporter-otlp: Dependencies required for OpenTelemetry PHP OTLP protocol data reporting.
open-telemetry/opentelemetry-auto-slim: Automatic event tracking packet of OpenTelemetry PHP for Slim framework implementation.
open-telemetry/opentelemetry-auto-pdo: Automatic event tracking packet of OpenTelemetry PHP for PHP DataObject implementation.

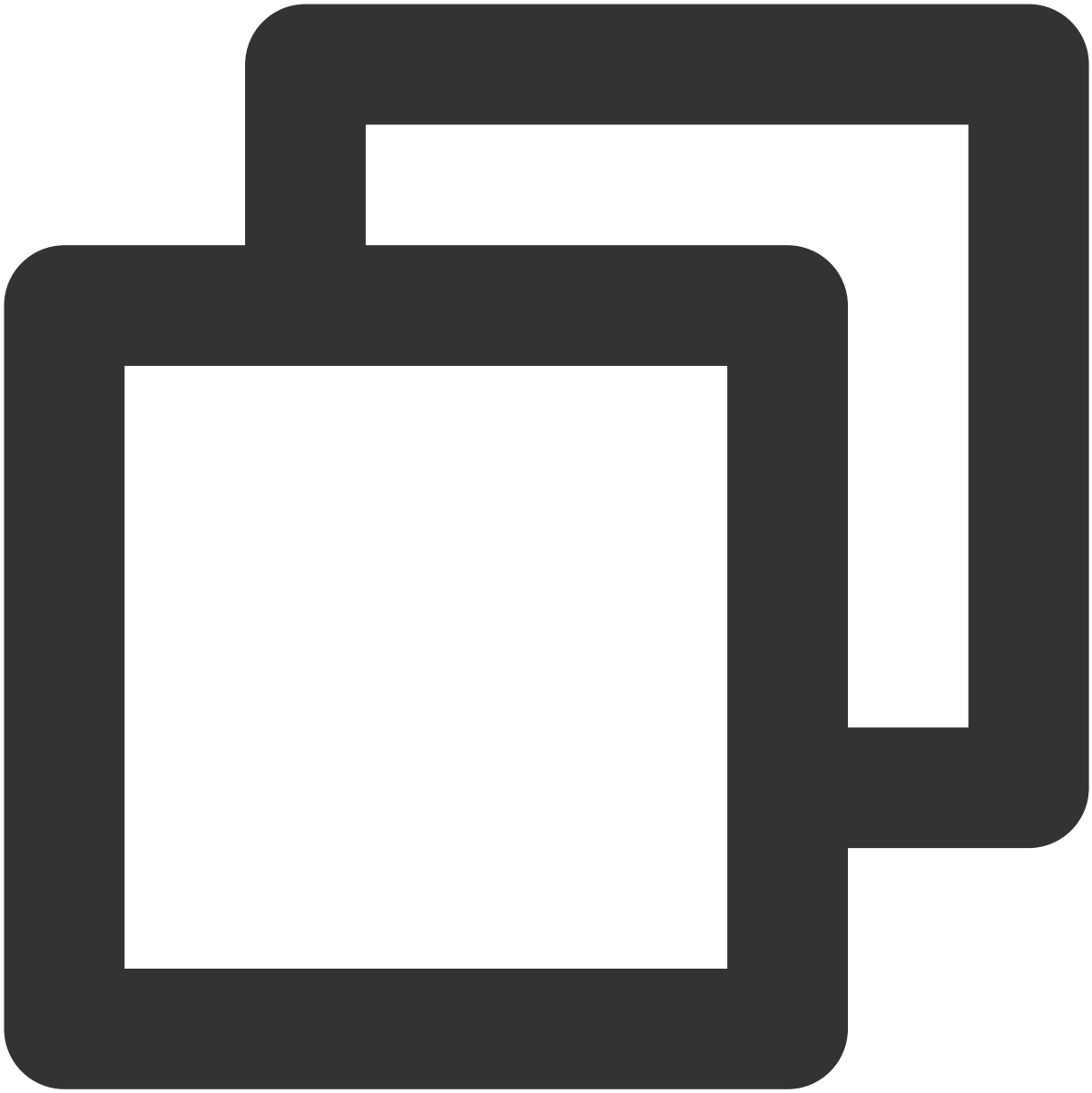
Note:

The packets `open-telemetry/opentelemetry-auto-slim` and `open-telemetry/opentelemetry-auto-pdo` are imported because the example demo uses the PDO and Slim frameworks. You can adjust according to your specific business needs. If your business components require OpenTelemetry automatic event tracking, you need to import the corresponding automatic event tracking packets into the project. For detailed import methods, see OpenTelemetry official documentation.

Step 2: Run the application.

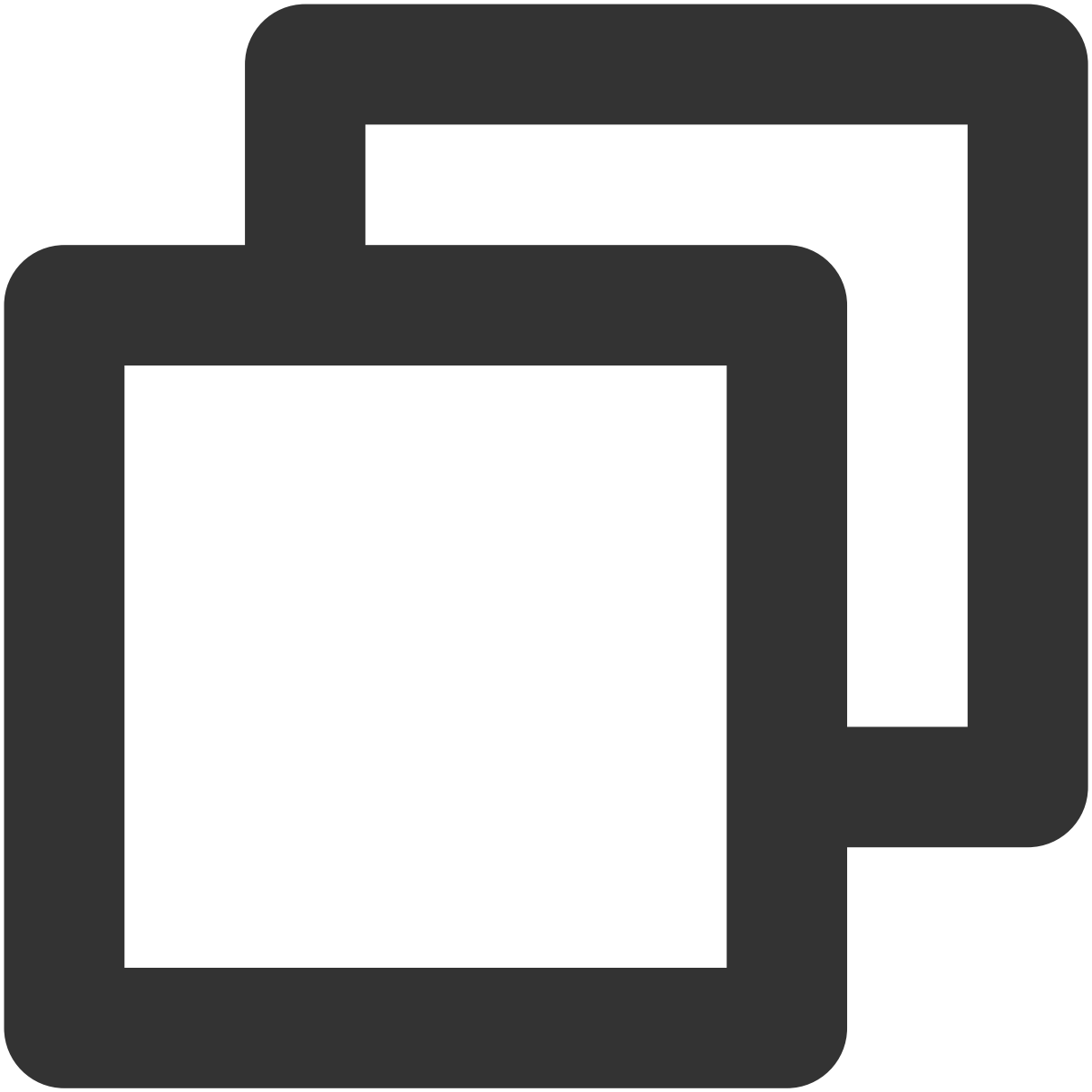
1. Execute the following command:





```
env OTEL_PHP_AUTOLOAD_ENABLED=true \\
 OTEL_TRACES_EXPORTER=otlp \\
 OTEL_METRICS_EXPORTER=none \\
 OTEL_LOGS_EXPORTER=none \\
 OTEL_EXPORTER_OTLP_PROTOCOL=grpc \\
 OTEL_EXPORTER_OTLP_ENDPOINT=<endpoint> \\ # Replace with the access point obtained
 OTEL_RESOURCE_ATTRIBUTES="service.name=<service-name>,token=<token>" \\ # Replace
 OTEL_PROPAGATORS=baggage,tracecontext \\
 php -S localhost:8080
```

2. Visit the following link in the browser:



```
http://localhost:8080/getID
```

Each time you visit this page, OpenTelemetry will automatically create a Trace and report the linkage data to APM.

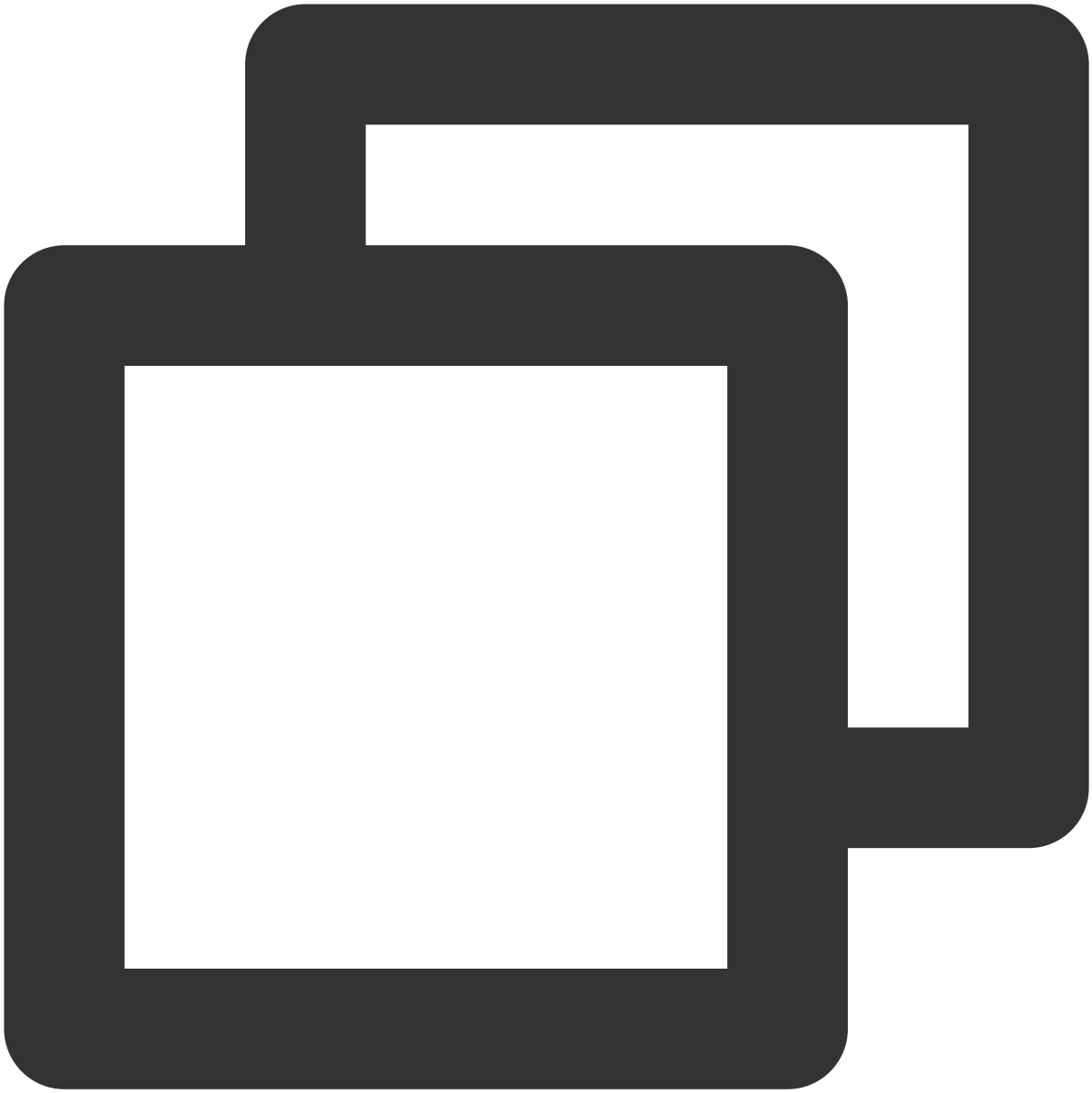
Connection Verification

After starting the PHP application, visit the corresponding interface via port 8080, for example, `https://localhost:8080/getID` . If there is normal traffic, the integrated application will be displayed in APM > Application monitoring > Application list, and the integrated application instances will be displayed in APM > Application monitoring > Application details > **Instance monitoring**. Since there is a certain delay in processing

observable data, if the application or instance is not found in the console after integration, please wait about 30 seconds.

Custom Event Tracking (Optional)

PHP Custom Tracing Documentation.' style="color:#000000;font-size:12px;"> When automatic event tracking does not meet your scenarios, or you need to add business layer event tracking, you can refer to the following content to use the OpenTelemetry PHP SDK to add custom event tracking. This document only demonstrates the most basic custom event tracking method. The OpenTelemetry community provides more flexible custom event tracking methods, and specific usage methods can be found in the OpenTelemetry community's PHP Custom Tracing Documentation.



```
<?php

use OpenTelemetry\\API\\Globals; // Required packet

require __DIR__ . '/vendor/autoload.php';
function wait(): void
{
    // Obtain the currently configured providers through the Globals packet.
    $tracerProvider = Globals::tracerProvider();
    $tracer = $tracerProvider->getTracer(
        'instrumentation-scope-name', //name (required)
```

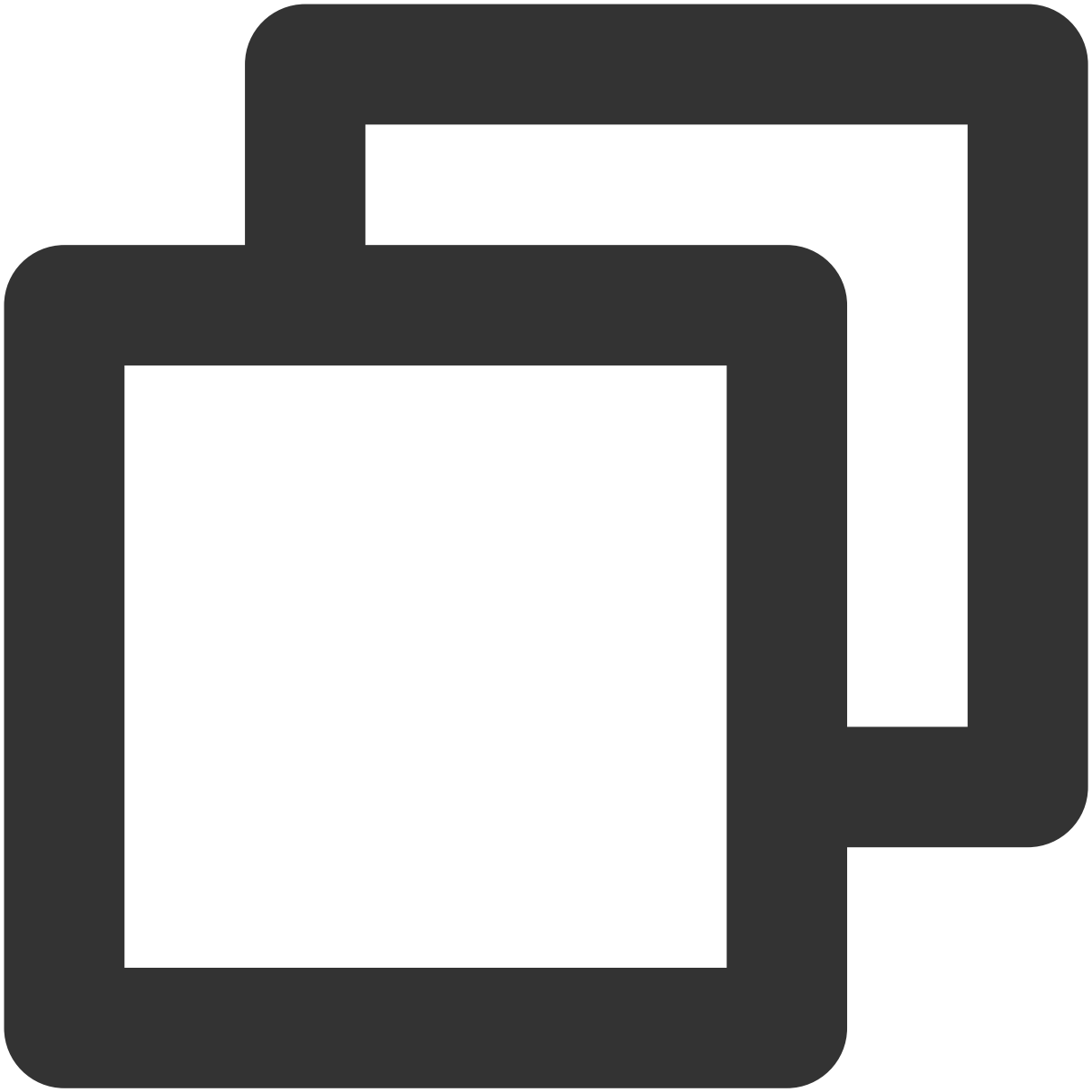
```
        'instrumentation-scope-version', //version
        'http://example.com/my-schema', //schema url
        ['foo' => 'bar'] //attributes
    );

    // Custom Event Tracking
    $span = $tracer->spanBuilder("wait")->startSpan();

    // Business code.null    sleep(5);null

    // Custom event tracking ends.
    $span->end();
}

wait();
```

Manual Connection Scheme

If your PHP application version cannot meet 8.0+ but can meet 7.4+, you can choose manual event tracking to report. This document only demonstrates the most basic manual event tracking method. The OpenTelemetry community provides more flexible manual event tracking methods, and specific usage methods can be found in the OpenTelemetry community's PHP Manual Integration Documentation.

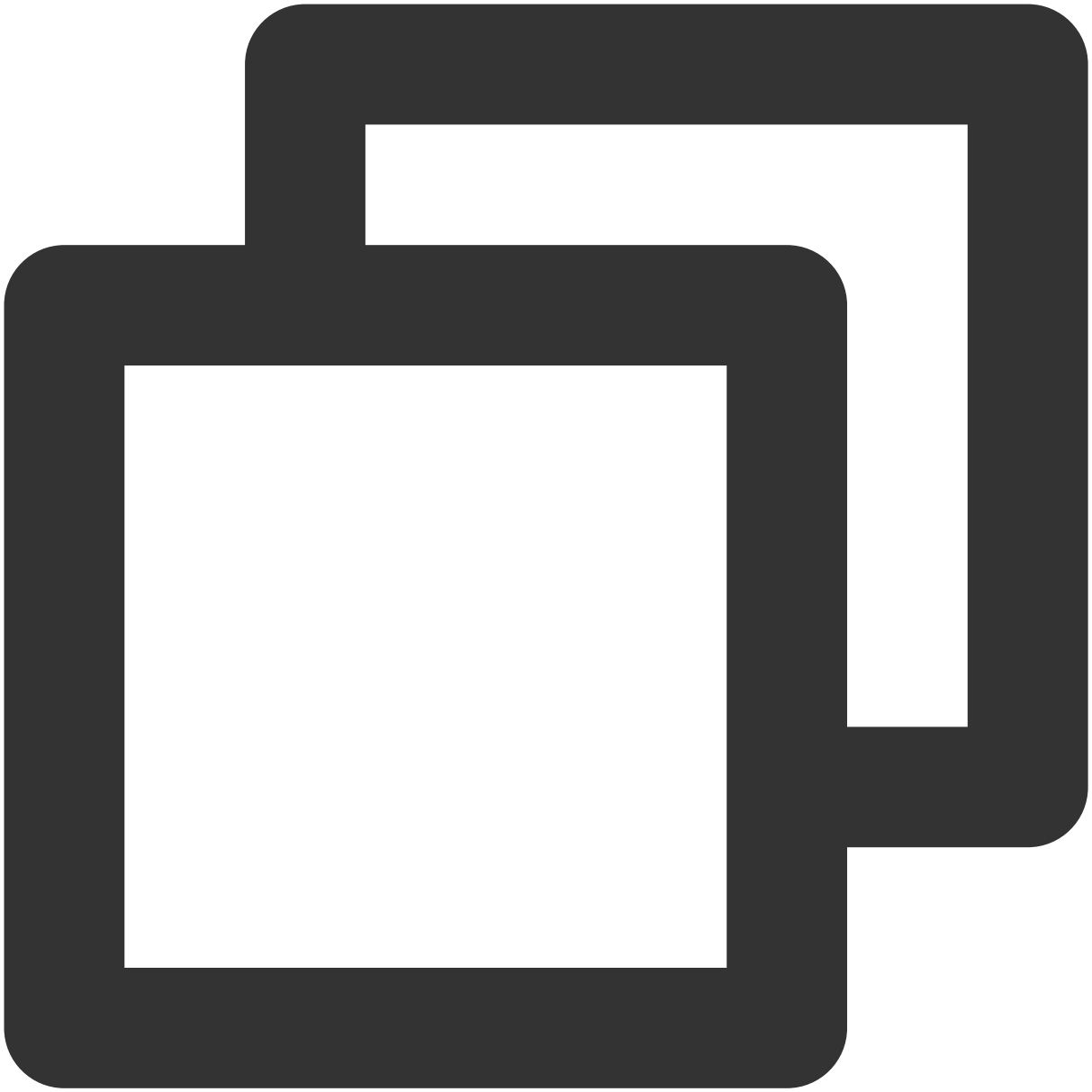
Import the dependencies needed for the OpenTelemetry PHP SDK and OpenTelemetry gRPC Explorer.

1. Download the PHP HTTP Client Library to report linkage data.



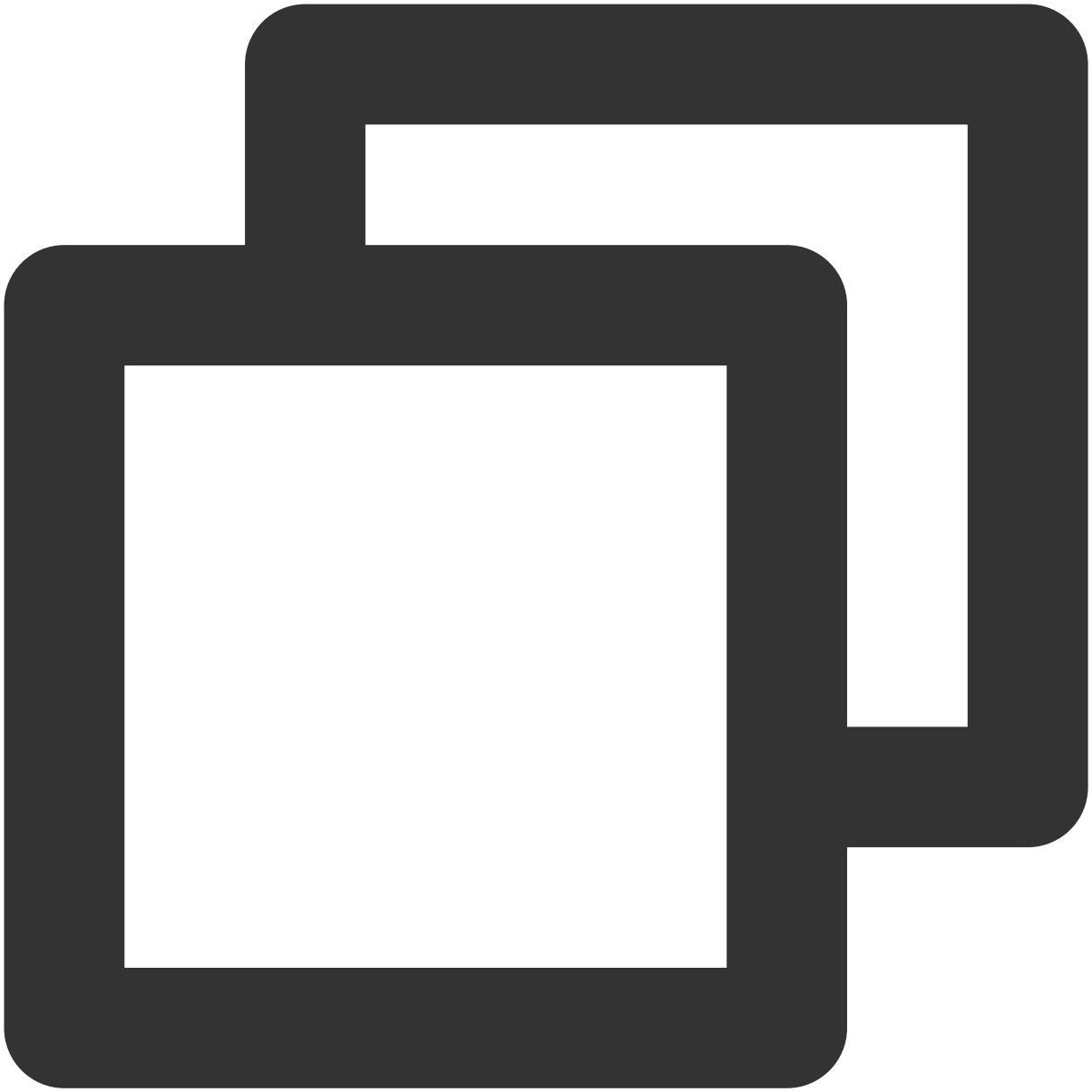
```
composer require guzzlehttp/guzzle
```

2. Download the OpenTelemetry PHP SDK.



```
composer require \\
 open-telemetry/sdk \\
 open-telemetry/exporter-otlp
```

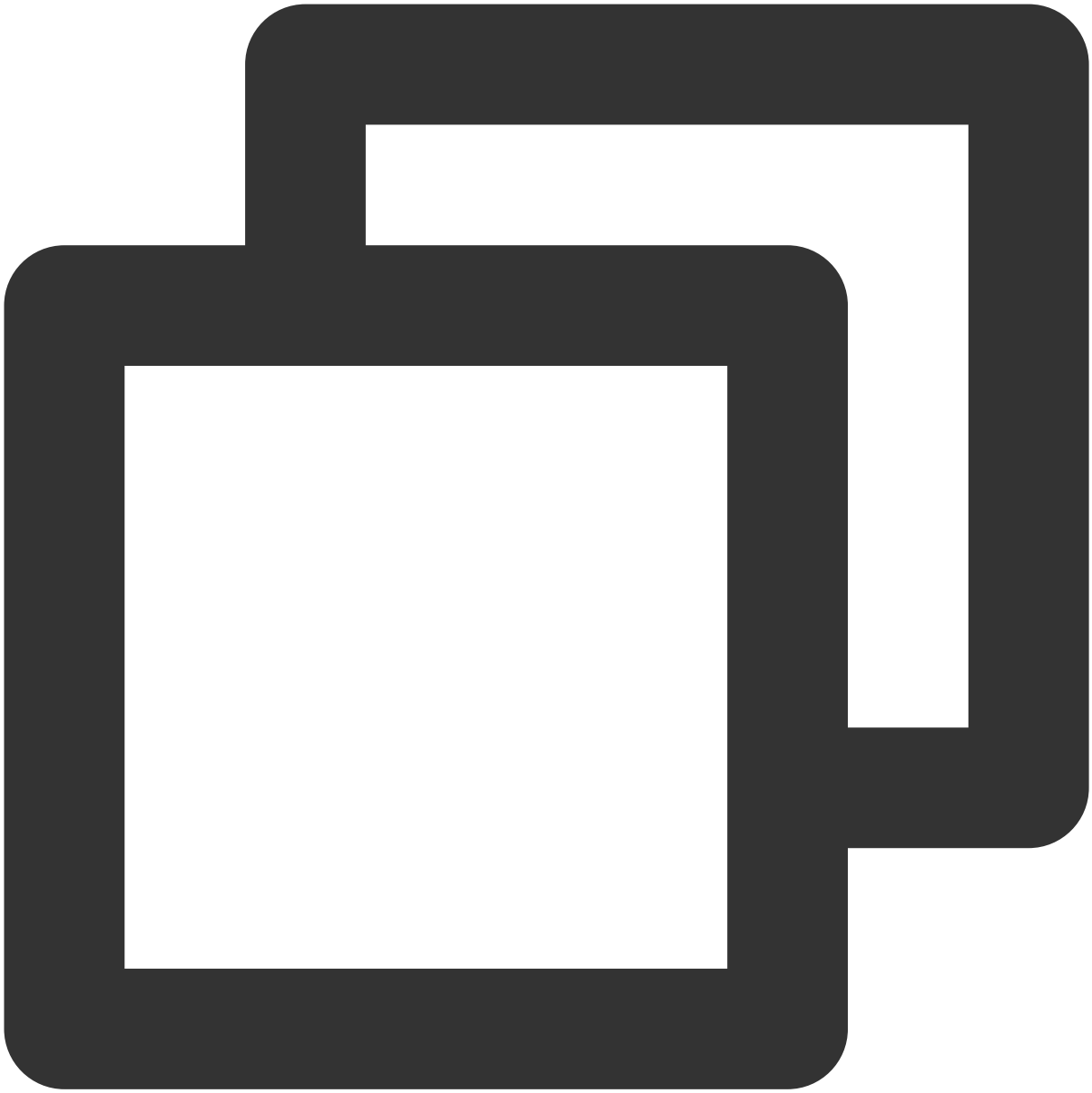
3. Download the dependencies needed to report data using gRPC.



```
pecl install grpc # Skip this step if gRPC has already been downloaded.
composer require open-telemetry/transport-grpc
```

Create an OpenTelemetry initialization tool.

Create the `opentelemetry_util.php` file in the directory where the `index.php` file is located and add the following code to the file:



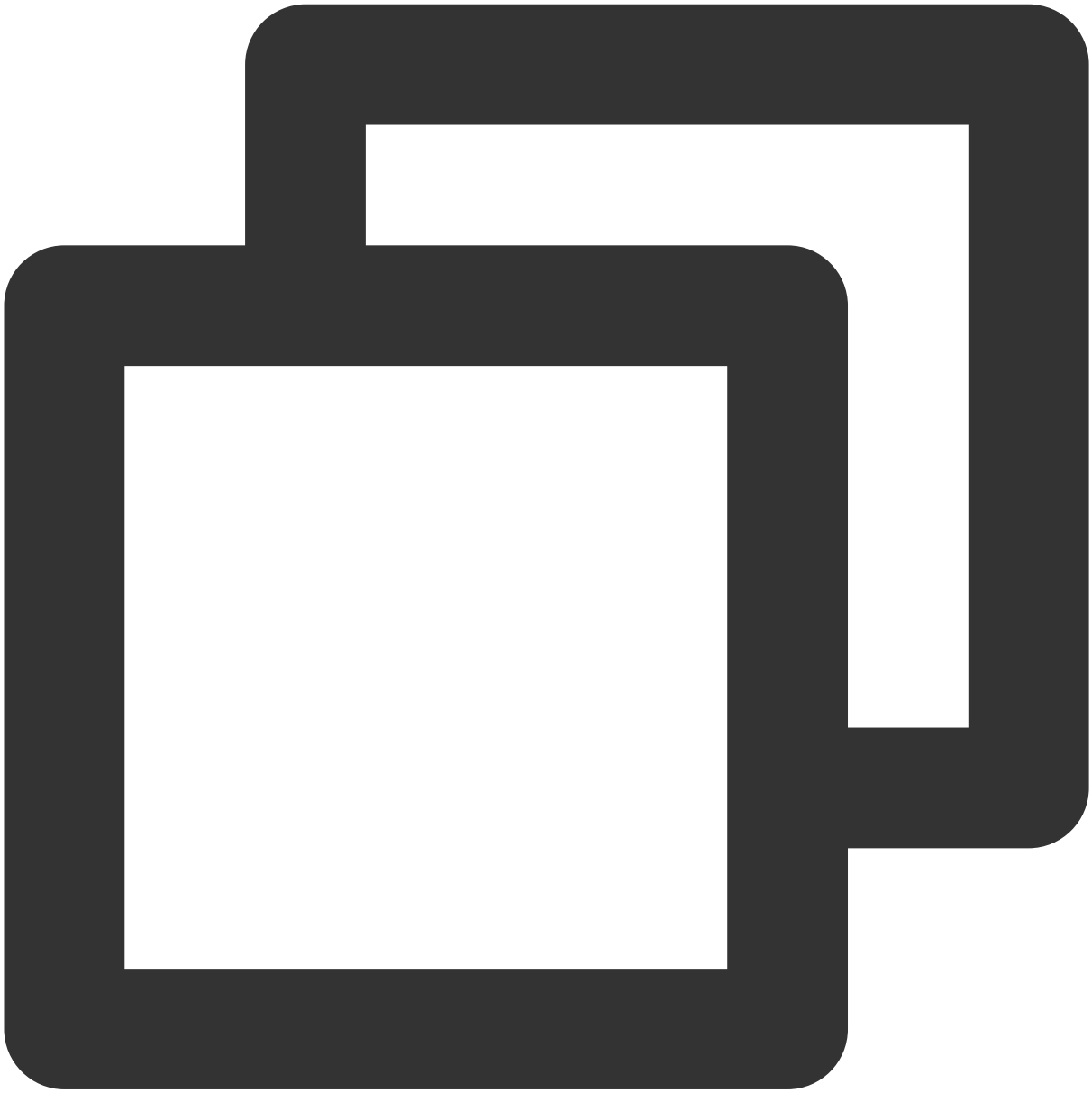
```
<?php
// Includes setting application name, Trace export method, Trace reporting access p

use OpenTelemetry\\API\\Globals;
use OpenTelemetry\\API\\Trace\\Propagation\\TraceContextPropagator;
use OpenTelemetry\\Contrib\\Otlp\\SpanExporter;
use OpenTelemetry\\SDK\\Common\\Attribute\\Attributes;
use OpenTelemetry\\SDK\\Common\\Export\\Stream\\StreamTransportFactory;
use OpenTelemetry\\SDK\\Resource\\ResourceInfo;
use OpenTelemetry\\SDK\\Resource\\ResourceInfoFactory;
use OpenTelemetry\\SDK\\Sdk;
```

```
use OpenTelemetry\\SDK\\Trace\\Sampler\\AlwaysOnSampler;
use OpenTelemetry\\SDK\\Trace\\Sampler\\ParentBased;
use OpenTelemetry\\SDK\\Trace\\SpanProcessor\\SimpleSpanProcessor;
use OpenTelemetry\\SDK\\Trace\\SpanProcessor\\BatchSpanProcessorBuilder;
use OpenTelemetry\\SDK\\Trace\\TracerProvider;
use OpenTelemetry\\SemConv\\ResourceAttributes;
use OpenTelemetry\\Contrib\\Grpc\\GrpcTransportFactory;
use OpenTelemetry\\Contrib\\Otlp\\OtlpUtil;
use OpenTelemetry\\API\\Signals;

// OpenTelemetry initialization configuration (OpenTelemetry initialization configu
function initOpenTelemetry()
{
  // 1. Set OpenTelemetry resource information.
  $resource = ResourceInfoFactory::emptyResource()->merge(ResourceInfo::create(Attr
  ResourceAttributes::SERVICE_NAME => '<your-service-name>', // Application name, r
  ResourceAttributes::HOST_NAME => '<your-host-name>' // hostname, optional.
  'token' => '<your-token>' // Replace with the token obtained in step 1.
  ])));


  // 2. Create a SpanExplorer to output spans to the console.
  // $spanExporter = new SpanExporter(
  // (new StreamTransportFactory())->create('php://stdout', 'application/json')
  // );

  // 2. Create a SpanExplorer to report spans via gRPC.
  $transport = (new GrpcTransportFactory())->create('<grpc-endpoint>' . OtlpUtil::m
  $spanExporter = new SpanExporter($transport);


  // 3. Create a global TraceProvider to create a tracer.
  $tracerProvider = TracerProvider::builder()
  ->addSpanProcessor(
  (new BatchSpanProcessorBuilder($spanExporter))->build()
  )
  ->setResource($resource)
  ->setSampler(new ParentBased(new AlwaysOnSampler()))
  ->build();

  Sdk::builder()
  ->setTracerProvider($tracerProvider)
  ->setPropagator(TraceContextPropagator::getInstance())
  ->setAutoShutdown(true) // Automatically shut down the tracerProvider after the P
  ->buildAndRegisterGlobal(); // Add the tracerProvider to the global.

}
```

```
?>
```

Modify the application code and create spans using the OpenTelemetry API.

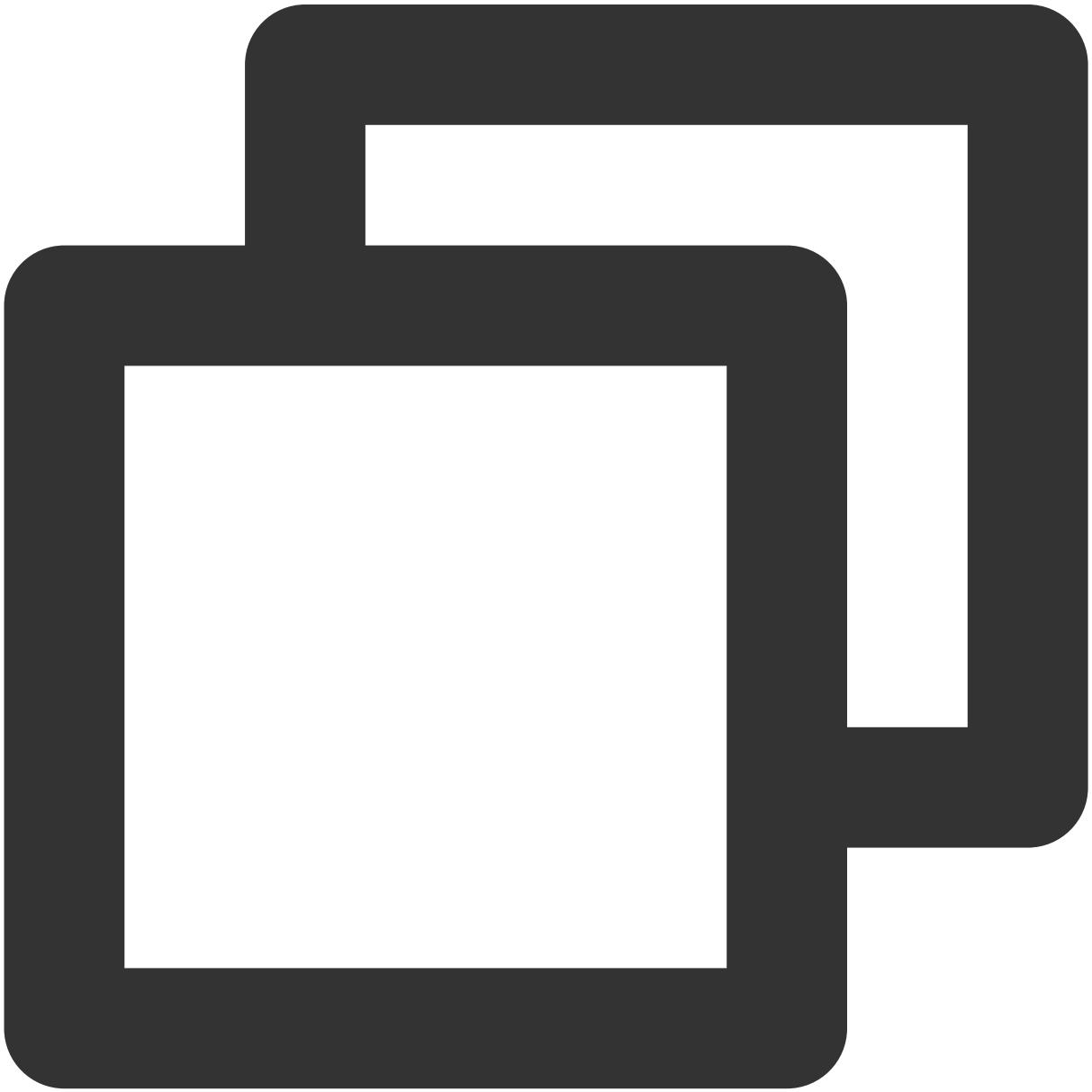
1. Import the required packets in the `index.php` file:



```
<?php

use OpenTelemetry\\API\\Globals;
use OpenTelemetry\\API\\Trace\\StatusCode;
use OpenTelemetry\\API\\Trace\\SpanKind;
```

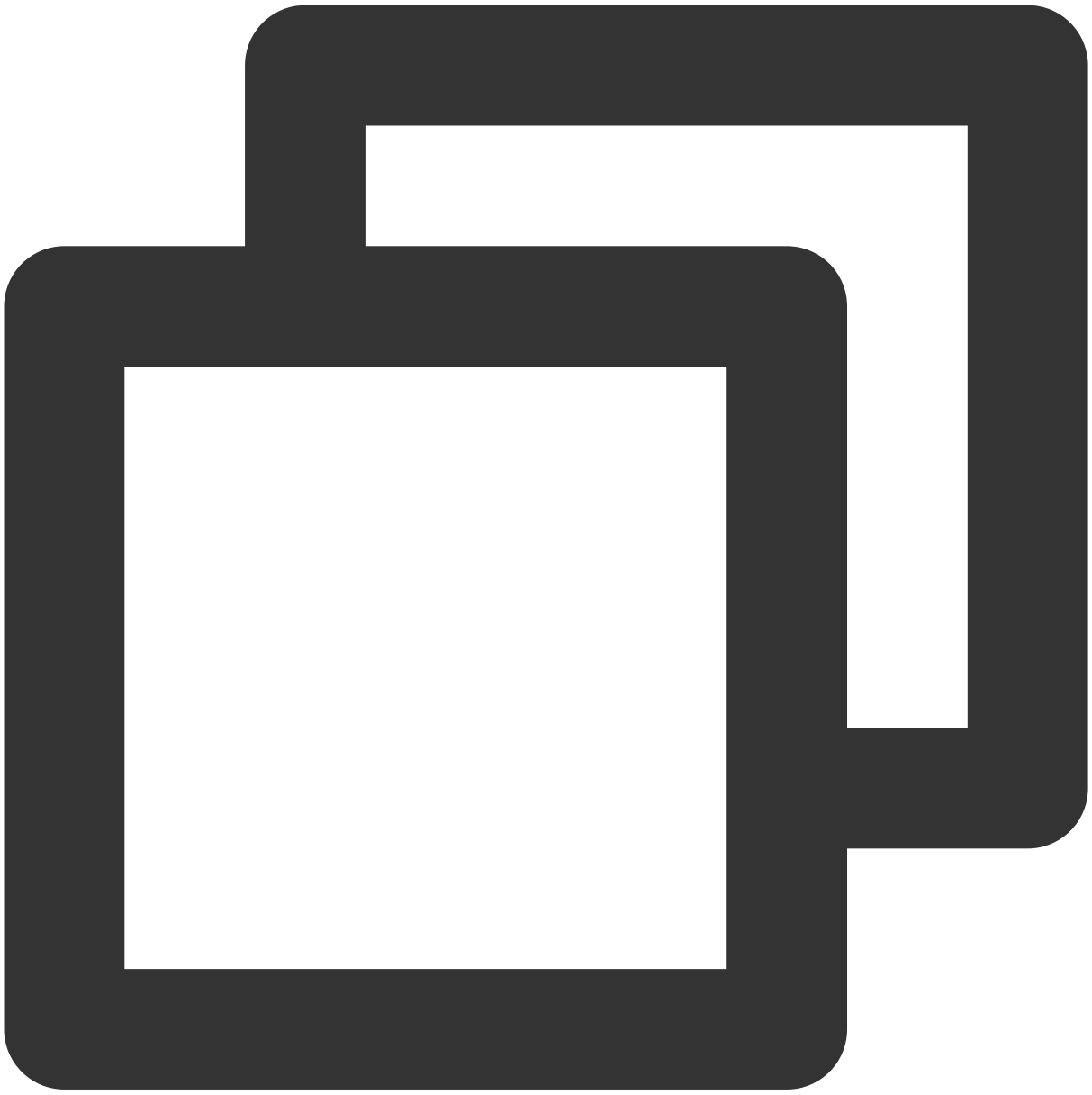
```
use OpenTelemetry\\SDK\\Common\\Attribute\\Attributes;
use OpenTelemetry\\SDK\\Trace\\TracerProvider;

use Psr\\Http\\Message\\ResponseInterface as Response;
use Psr\\Http\\Message\\ServerRequestInterface as Request;
use Slim\\Factory\\AppFactory;

require __DIR__ . '/opentelemetry_util.php';
```

2. Call the `initOpenTelemetry` method to complete initialization. OpenTelemetry initialization configuration needs to be done when the PHP application initializes:



```
// OpenTelemetry initialization, including setting the application name, trace expo
initOpenTelemetry();
```

3. Create a span in the `rolldice` API.

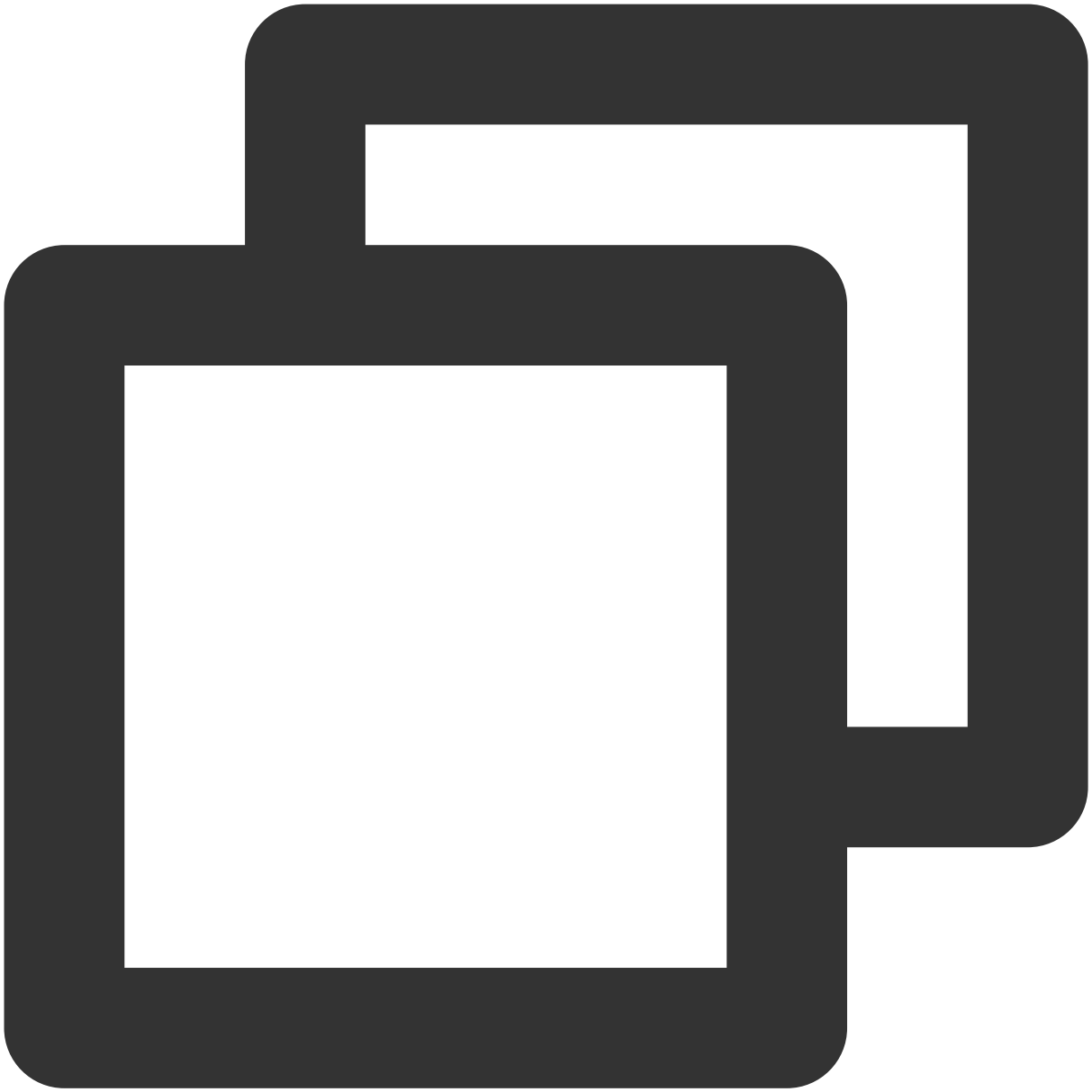


```
/**
 * 1. API feature: Simulate rolling a dice, returning a random integer between 1 an
 * Demonstrate how to create a span, set attributes, events, and events with attrib
 */
$app->get('/rolldice', function (Request $request, Response $response) {
 // Obtain tracer.
```

```
  $tracer = \\OpenTelemetry\\API\\Globals::tracerProvider()->getTracer('my-tracer');
  // Create span; set span kind; default is KIND_INTERNAL if not set.
  $span = $tracer->spanBuilder("/rolldice")->setSpanKind(SpanKind::KIND_SERVER)->sta
  // Set attributes for span.
  $span->setAttribute("http.method", "GET");
  // Set events for span.
  $span->addEvent("Init");
  // Set events with attributes.
  $eventAttributes = Attributes::create([
  "key1" => "value",
  "key2" => 3.14159,
  ]);

  // Business code.
  $result = random_int(1,6);
  $response->getBody()->write(strval($result));

  $span->addEvent("End");
  // Terminate span.
  $span->end();

  return $response;
 });
```

4. Create nested span.

Create a `rolltwodices` API to simulate rolling two dice, returning two random positive integers between 1 and 6. The following code demonstrates how to create nested spans:

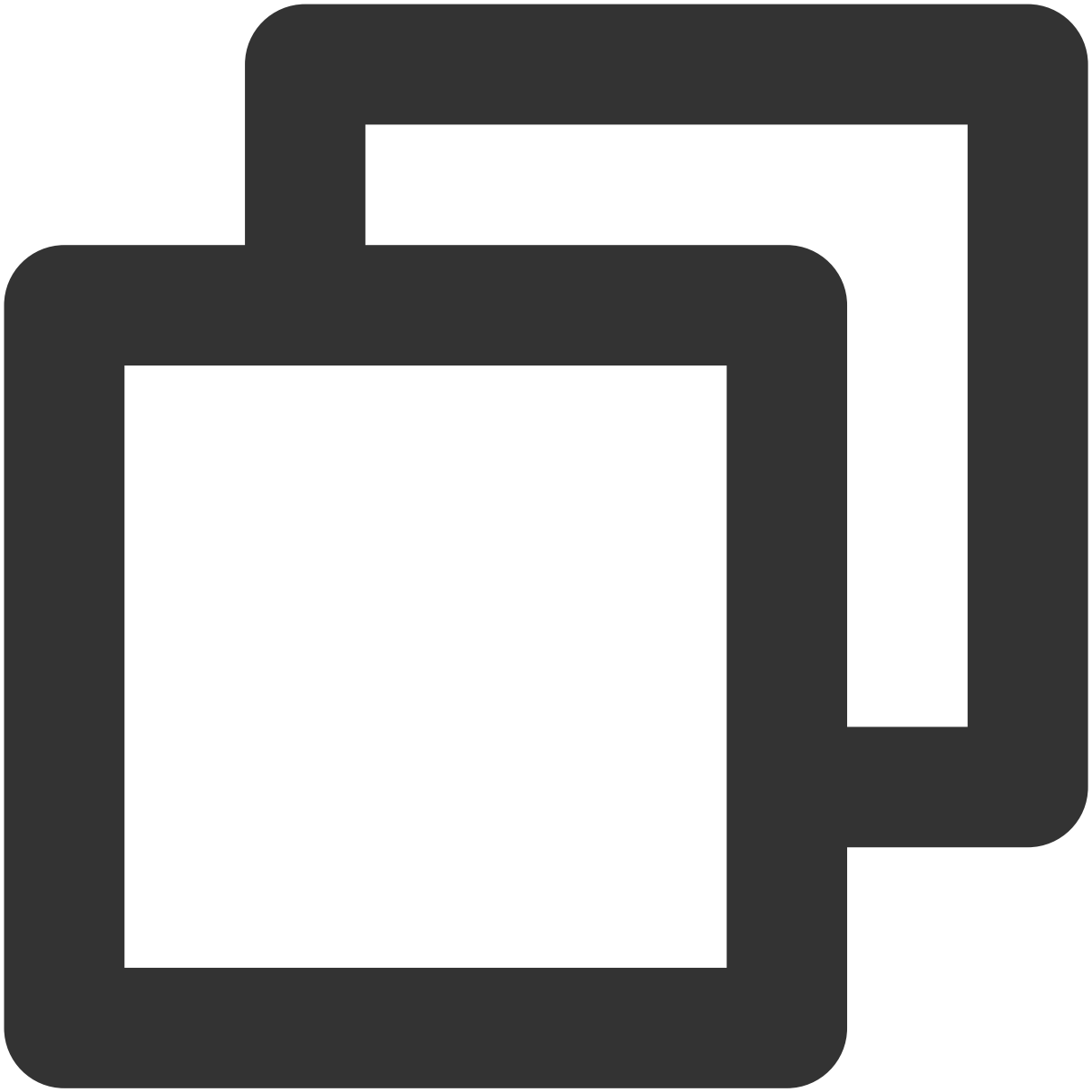


```
$app->get('/rolltwodices', function (Request $request, Response $response) {
 // Obtain tracer.
 $tracer = \\OpenTelemetry\\API\\Globals::tracerProvider()->getTracer('my-tracer');
 // Create span.
 $parentSpan = $tracer->spanBuilder("/rolltwodices/parent")->setSpanKind(SpanKind::
 $scope = $parentSpan->activate();

 $value1 = random_int(1,6);

 $childSpan = $tracer->spanBuilder("/rolltwodices/parent/child")->startSpan();
```

```
  // Business code.
  $value2 = random_int(1,6);
  $result = "dice1: " . $value1 . ", dice2: " . $value2;

  // Terminate span.
  $childSpan->end();
  $parentSpan->end();
  $scope->detach();

  $response->getBody()->write(strval($result));
  return $response;
});
```

5. Use span to record exceptions that occur in the code.

Create an `error` API to simulate an API exception. The following code demonstrates how to use span to record the status when an exception occurs in the code:

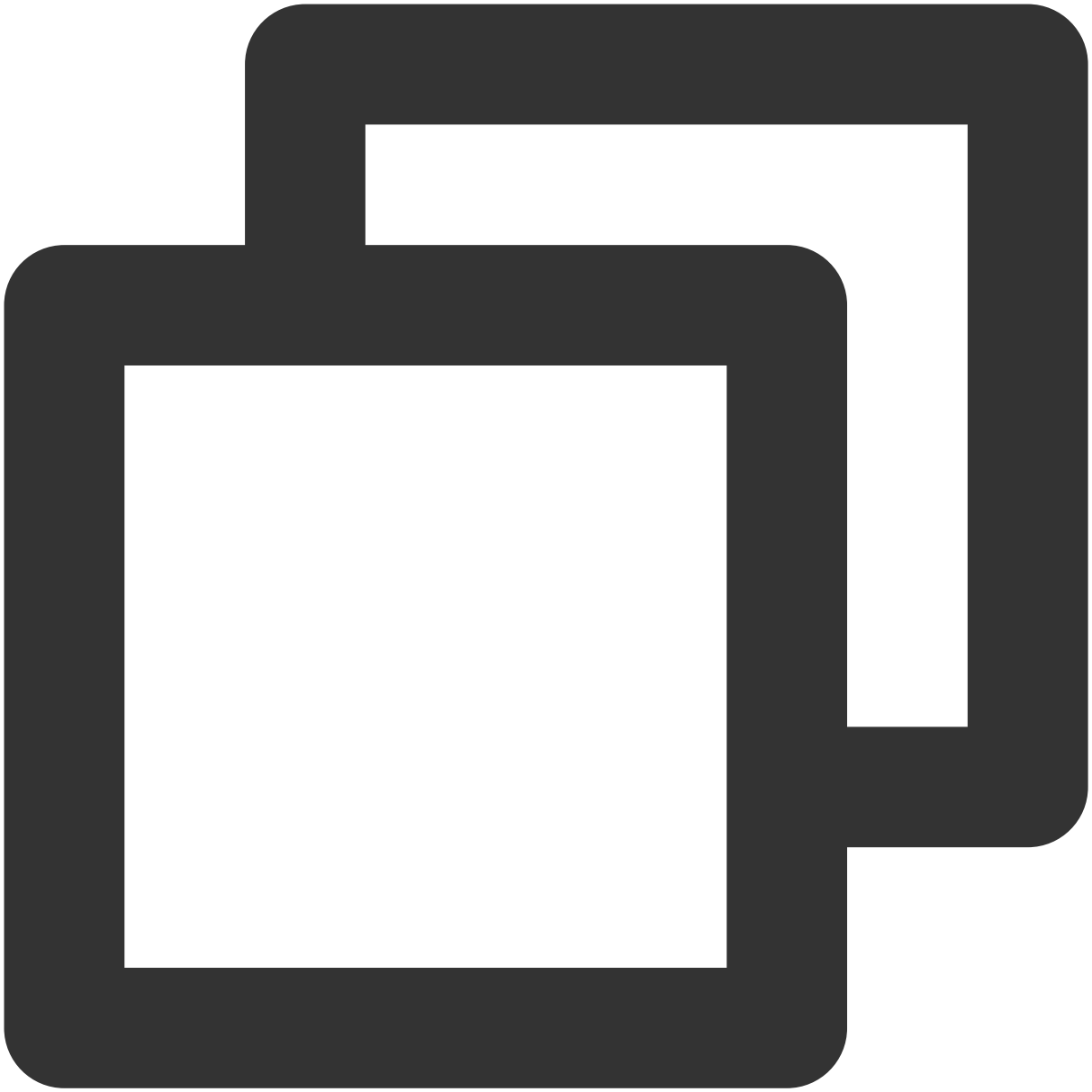


```
$app->get('/error', function (Request $request, Response $response) {
 // Obtain tracer.
 $tracer = \\OpenTelemetry\\API\\Globals::tracerProvider()->getTracer('my-tracer');
 // Create span.
 $span3 = $tracer->spanBuilder("/error")->setSpanKind(SpanKind::KIND_SERVER)->start
 try {
 // Simulate code exception.
 throw new \\Exception('exception!');
 } catch (\\Throwable $t) {
 // Set span status to error.
 $span3->setStatus(\\OpenTelemetry\\API\\Trace\\StatusCode::STATUS_ERROR, "expcetio
```

```
  // Record exception stack track.
  $span3->recordException($t, ['exception.escaped' => true]);
  } finally {
  $span3->end();
  $response->getBody()->write("error");
  return $response;
  }
});
```

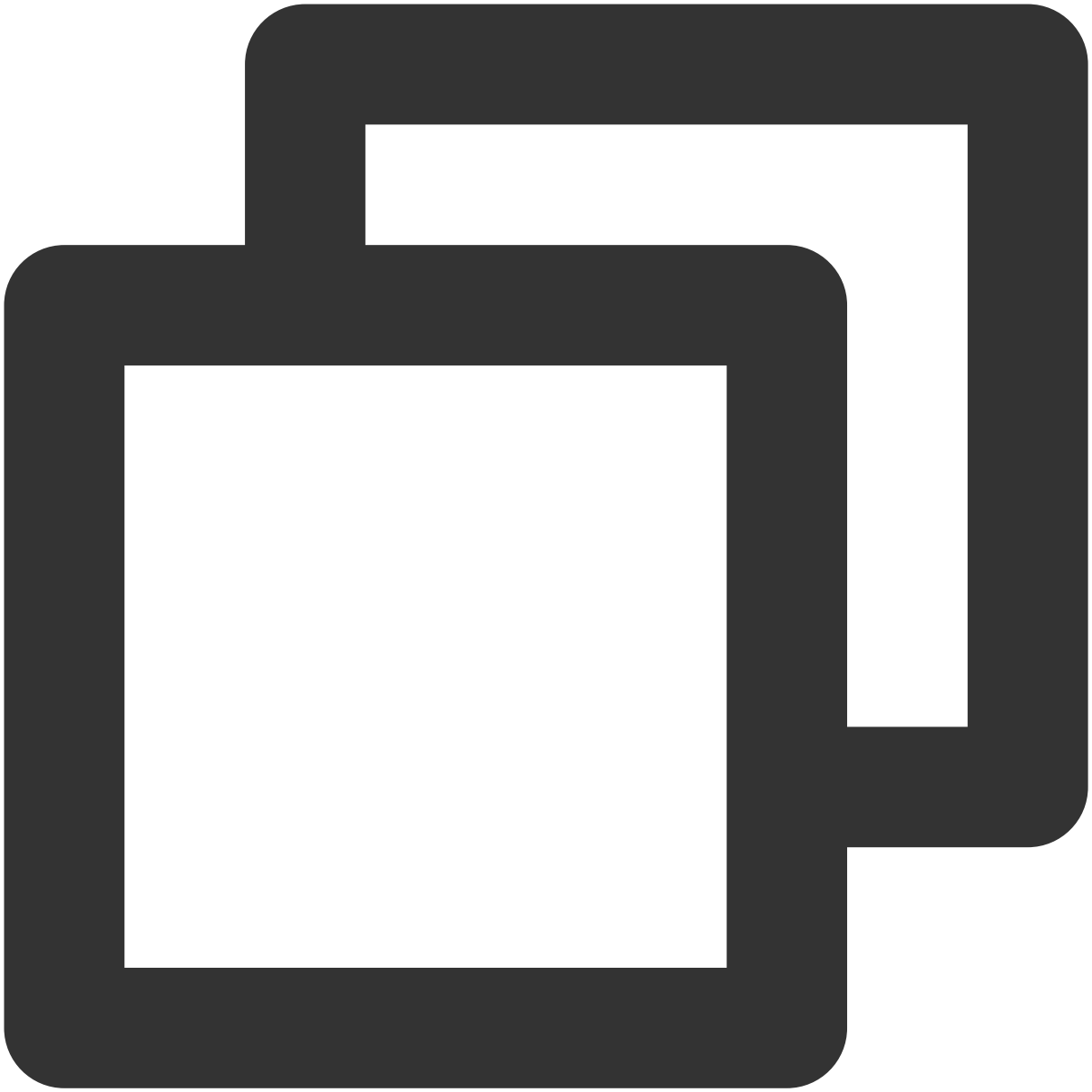
Run the application.

1. Execute the following command:



```
php -S localhost:8080
```

2. Visit the following link in the browser:



```
http://localhost:8080/rolldice
http://localhost:8080/rolltwodices
http://localhost:8080/error
```

Each time the page is accessed, OpenTelemetry will create linkage data and report it to APM.

Connection Verification

After starting the PHP application, visit the corresponding interface via port 8080, for example, `https://localhost:8080/getID` . If there is normal traffic, the integrated application will be displayed in APM

> Application monitoring > Application list, and the integrated application instances will be displayed in APM > Application monitoring > Application details > **Instance monitoring**. Since there is a certain delay in processing observable data, if the application or instance is not found in the console after integration, please wait about 30 seconds.

Installing tencent-opentelemetry-operator

Last updated：2024-06-19 16:31:30

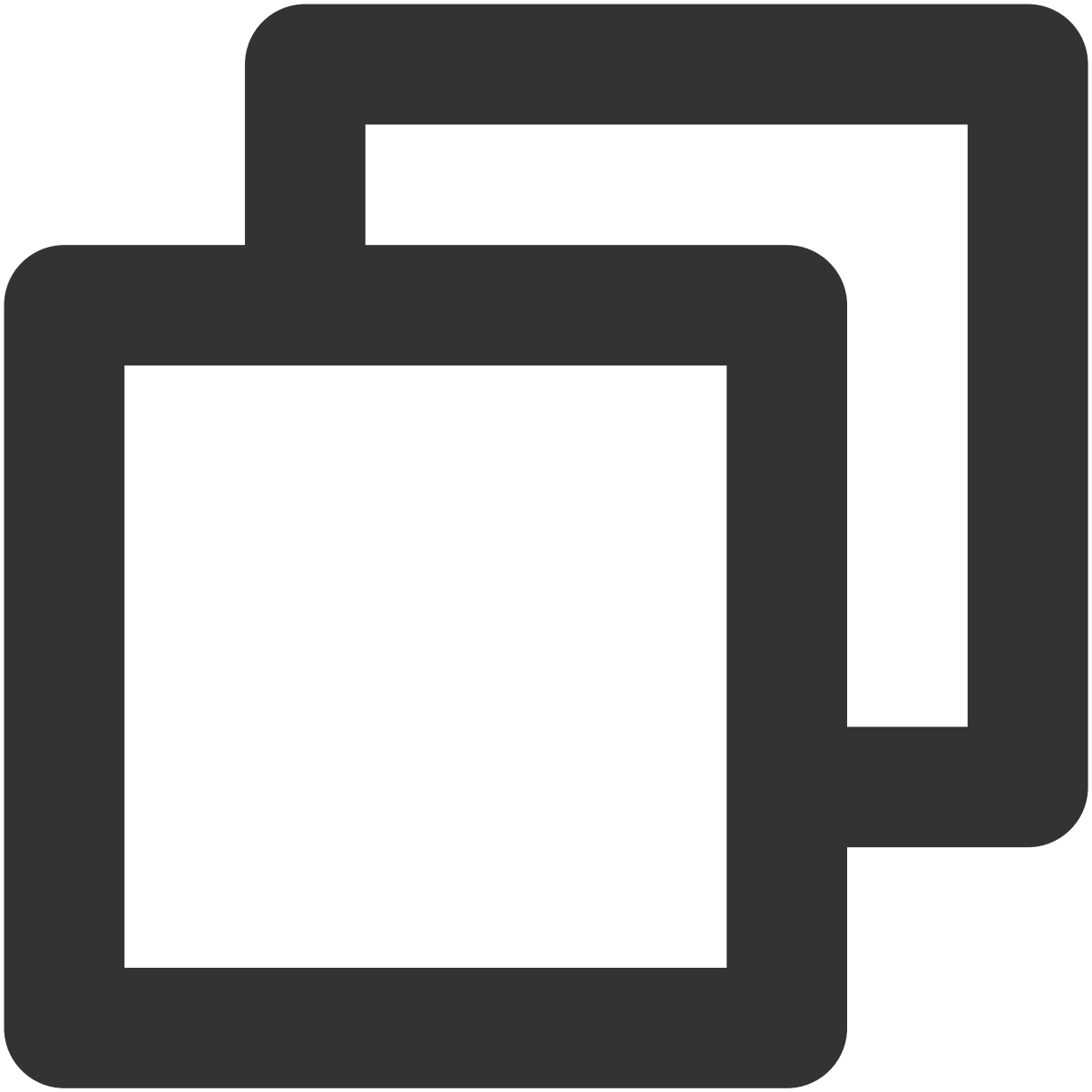
For applications deployed on TKE, the Tencent Cloud observability team offers an Operator solution: tencent-opentelemetry-operator. Built upon the community opentelemetry-operator, it enables agent automatic injection, facilitating applications connecting APM quickly. Currently, tencent-opentelemetry-operator supports programming languages including Java, Python, Node.js, and .Net.

Note:

tencent-opentelemetry-operator supports Kubernetes version 1.19 and above for TKE standard clusters and TKE Serverless clusters but does not support edge clusters and register clusters.

Configuration Items Description

tencent-opentelemetry-operator is deployed via Helm, with all configuration items centralized in `values.yaml`. Pay attention to the hierarchical relationship of parameters within the YAML file. See the following YAML snippet:



```
env:
  TKE_CLUSTER_ID: "cls-ky8nmlra"
  TKE_REGION: "ap-guangzhou"
  APM_ENDPOINT: "http://pl.ap-guangzhou.apm.tencentcs.com:4317"
  APM_TOKEN: "apmdemotoken"
```

Required Field

Parameter	Description

env.TKE_CLUSTER_ID	TKE cluster ID.
env.TKE_REGION	TKE Cluster region, for example, ap-guangzhou. For more details, see CVM Regions and AZs value range.
env.ENDPOINT	APM private network connect point. Each cluster must specify a unique APM **private** network connect point.
env.APM_TOKEN	Default APM business system token, which can specify other business systems at the workload level.

Optional Field

Parameter	Description
env.JAVA_INSTR_VERSION	Java agent version. You may fill in `latest` (default) or a specific version number. Filling in this field is not recommended unless necessary.
env.PYTHON_INSTR_VERSION	Python agent version. You may fill in `latest` (default) or a specific version number. Filling in this field is not recommended unless necessary.
env.NODEJS_INSTR_VERSION	Node.js agent version. You may fill in `latest` (default) or a specific version number. Filling in this field is not recommended unless necessary.
env.DOTNET_INSTR_VERSION	.Net agent version. You may fill in `latest` (default) or a specific version. Filling in this field is not recommended unless necessary.
env.INTL_SITE	For the international site, fill in `true` .

Note:
If you need to specify a specific version number of the Agent, go to Agent Version Information to get the version number.

Installing Method

One-click Installation via the APM Console (Recommended)

Due to the complexity of filling configuration items, it is recommended to use the one-click installation of the tencent-opentelemetry-operator feature via the APM console to simplify the installation steps.
1. Log in to the TCOP console.
2. In the left menu column, select **Application Performance Management > Application monitoring**, and click **Application list > Access application**.

3. Click on the language you need to connect, and select **Automatic onboarding of TKE environment** as the reporting method.
4. Click **One-click Install Operator**.
5. In the pop-up dialog box, select the corresponding reporting region, default business system, TKE's region, and TKE cluster, and click **Confirm** to complete the installation in the corresponding TKE cluster.

Note:
The tencent-opentelemetry-operator one-click installed via in the APM console will be installed in the kube-system namespace. If you need to modify related configuration items, you can update the same TKE cluster through the console.

Installing via TKE Application Market

1. Log in to TKE console.
2. In the left sidebar, select **Application Market** and search for tencent-opentelemetry-operator.
3. Click **Create Application**, select the TKE cluster you want to install, and fill in the required parameters to complete the installation.

Note:
The tencent-opentelemetry-operator installed via the TKE application market can be installed in any namespace. In the same TKE cluster, only one tencent-opentelemetry-operator can be installed at most.

Connecting Applications

After the installation of tencent-opentelemetry-operator, the `opentelemetry-operator-system` namespace is automatically created, and related Kubernetes resources are created. By adding related annotations to the workloads that need to connect APM, agent automatic injection can be realized, and monitoring data can be reported to APM.

Upgrading Agent Version

Last updated：2024-06-19 16:31:30

Automatic Connection Mode for TKE Environment

Applications that automatically connect to the TKE environment use the agent auto-update policy by default. Users do not need to worry about agent upgrades. Before they push a new version of the agent, the APM team conducts multiple rounds of stability testing to ensure the agent's stability and compatibility.
If you have already actively specified an agent version at the Operator or workload level, it is recommended to remove the configuration items used to specify the agent version and return to the auto-update mode. If it is indeed necessary to specify a specific agent version, see Agent Version Information to obtain the updated version number.

Non-Automatic Connection Mode for TKE Environment

See directions in respective connection documentation, download and update the agent packet, or introduce dependencies of updated versions.
If you are using the Tencent Cloud OpenTelemetry Java Agent Enhanced Edition, you can go to Agent Version Information to download new versions of the agent.