

Free, Intelligent and Serverless Page View/Read Count using Google Analytics and Cloudflare Workers

Using Google Analytics to generate a SVG image of the Page Reader/View Count through Cloudflare Workers

Kabui, Charles

2024-11-23

Table of Contents

Authenticating and Authorizing Google Analytics APIs	3
Google APIs Explorer	4
Python Example	8
Code	8
Tests	12
Using Cloudflare Worker	13
Implementing Typescript Worker	13
Testing, Deployment and Troubleshooting Cloudflare Workers	20
Alternatives to Cloudflare	21

[Read at ToKnow.ai](#)

[k Open in Kaggle](#)

[Download as Notebook](#)



When you set up a website, blog, vlog, or social media post, you expect engagement. One of the simplest and most effective ways to measure engagement is through views. View count is effective because it measures both active and passive audience interaction. However, not all views represent actual engagement. For example, if one user reloads the same page a thousand times, it doesn't represent a thousand unique viewers - it's just one user refreshing repeatedly. They may not have even read the content. On the other hand, when a user returns multiple times to read content carefully, each visit could be considered meaningful engagement. To calculate actual reads, we need some intelligence in our tracking.

In the case of a blog or website, one of the simplest ways to add some intelligence is to delay sending the view or read signal by for example 7 seconds. Another way is to wait for the user to scroll a page to about half the page before triggering a read count.

You could also employ a free external service such as <https://visitorbadge.io>, which receives the page url and returns an SVG image of the page count, for example:

<https://api.visitorbadge.io/api/visitors?path=https://toknow.ai/posts/display-google-analytics-views-using-cloudflare-worker/index.html>

One of the drawback of this strategy is we can't delay the trigger of view count and show the current views/reads at the same time.

Ofcourse, you can implement a backend service to save and read the views. However, for a static websites and blogs such as this one (hosted as github pages), that would go against the current architecture.

Another alternative would be to utilize Google analytics. Google analytics can be added not just for the views, but to measure the overall health of the website/blog. If google analytics is your primary analytics tool for your website/blog, why not just show the views they have calculated?

To ensure we dont affect the current architecture, we can use a cloudflare worker to fetch views from google analytics.

i Note

[Google Analytics](#) helps website and app owners track and analyze their website traffic. It gives insights into how users interact with your website, such as which pages they visit, how long they stay, and where they come from, demographics, page views, session duration, and more. To set up Google Analytics, create a Google account, set up a property for your website, and add the provided tracking code to your website's HTML. For a step-by-step guide on how to set up Google Analytics, read <https://dev.to/codesphere/setting-up-google-analytics-on-your-static-website-47mn> and <https://morotsman.github.io/blog,/google/analytics,/jekyll,/github/pages/2020/07/07/add-google-analytics.html> or watch the video below:

Click to watch the video at Youtube.



Authenticating and Authorizing Google Analytics APIs

! Important

This assumes you already have Google Analytics setup and running on one of your website/blog

For a detailed guide on how to create a service account, download the `credentials.json` file, and connect the service account to the google analytics account, see <https://developers.google.com/analytics/devguides/reporting/data/v1/quickstart-client-libraries>.

The `credentials.json` file has the following json structure:

```
{  
  "type": "service_account",  
  "project_id": "...",  
  "private_key_id": "s6sdfsdf876...sdf7ygsf78fsd",
```

```
"private_key": "-----BEGIN PRIVATE KEY-----\n...\n-----END PRIVATE KEY-----\n",
"client_email": "starting-account...iam.gserviceaccount.com",
"client_id": "094534589...438598743",
"auth_uri": "https://accounts.google.com/o/oauth2/auth",
"token_uri": "https://oauth2.googleapis.com/token",
"auth_provider_x509_cert_url": "https://www.googleapis.com/oauth2/v1/certs",
"client_x509_cert_url": "https://www.googleapis.com/robot/v1/metadata/x509/starting-account",
"universe_domain": "googleapis.com"
}
```

To get the Google Analytics property identifier, see https://developers.google.com/analytics/devguides/reporting/data/v1/property-id#what_is_my_property_id

We need both the Google Analytics property identifier (`properties/232..132`) and `credentials.json` file to be able to fetch analytics data.

Google APIs Explorer

Google provides an APIs Explorer specifically for testing Google Analytics Data API calls.

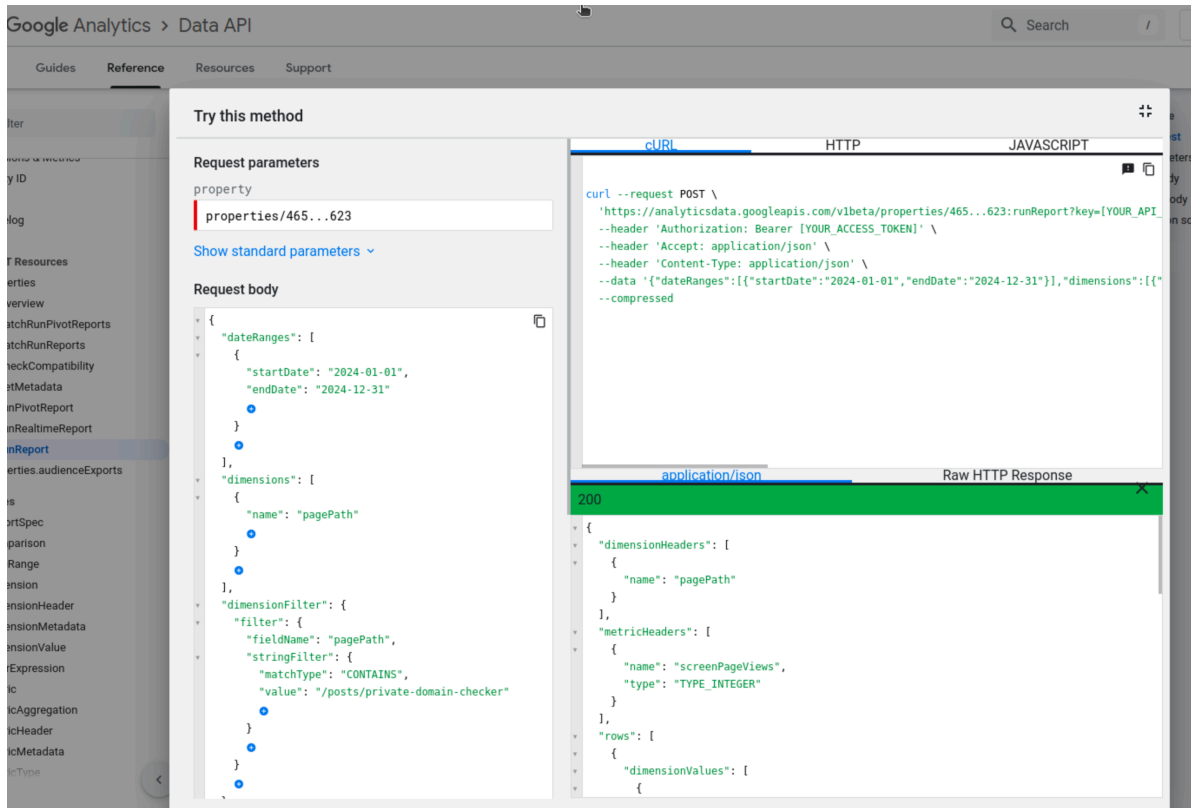


Figure 1: Google Analytics API Explorer

Here's how to use it:

- Visit: <https://developers.google.com/analytics/devguides/reporting/data/v1/rest/v1beta/properties/runReport>
- Replace `PROPERTY_ID` with your actual Google Analytics 4 property ID, instructions here: <https://developers.google.com/analytics/devguides/reporting/data/v1/property-id>
- Use the following request body and update the `dateRanges.startDate` (format: YYYY-MM-DD), `dateRanges.endDate` (format: YYYY-MM-DD) and page path (`dimensionFilter.filter.stringFilter.value`) whose views you want to get. The `startDate` can be any old day, even older than the website and `endDate` can be any future date.

```

{
  "dateRanges": [
    {
      "startDate": "2024-01-01",
      "endDate": "2024-12-31"
    }
  ]
}

```

```

    }
  ],
  "dimensions": [
    {
      "name": "pagePath"
    }
  ],
  "dimensionFilter": {
    "filter": {
      "fieldName": "pagePath",
      "stringFilter": {
        "matchType": "CONTAINS",
        "value": "/posts/private-domain-checker"
      }
    }
  },
  "metrics": [
    {
      "name": "screenPageViews"
    }
  ]
}

```

- Click the “**Execute**” button.

This will allow you to send a test POST request with your configured parameters and see the response from the API, including any errors or the data you requested.

Success Response:

```

// HTTP/1.1 200
{
  "dimensionHeaders": [
    {
      "name": "pagePath"
    }
  ],
  "metricHeaders": [
    {
      "name": "screenPageViews",
      "type": "TYPE_INTEGER"
    }
  ],
}

```

```
"rows": [
  {
    "dimensionValues": [
      {
        "value": "/posts/private-domain-checker/"
      }
    ],
    "metricValues": [
      {
        "value": "82"
      }
    ]
  },
  {
    "dimensionValues": [
      {
        "value": "/posts/private-domain-checker/index.html"
      }
    ],
    "metricValues": [
      {
        "value": "5"
      }
    ]
  },
  {
    "dimensionValues": [
      {
        "value": "/posts/private-domain-checker"
      }
    ],
    "metricValues": [
      {
        "value": "2"
      }
    ]
  }
],
"rowCount": 3,
"metadata": {
  "currencyCode": "USD",
  "timeZone": "Africa/Nairobi"
}
```

```
},  
  "kind": "analyticsData#runReport"  
}
```

Error Reponse:

```
// HTTP/1.1 400  
{  
  "error": {  
    "code": 400,  
    "message": "Invalid property ID: 465...623. A numeric Property ID is required.  
    To learn more about Property ID, see  
    https://developers.google.com/analytics/devguides/reporting/data/v1/property-id.",  
    "status": "INVALID_ARGUMENT"  
  }  
}
```

Python Example

install the following packages, [google-auth-oauthlib](#) and [google-api-python-client](#) for ease of authentication. You need to obtain a `property_key` and credentials from google. You also need to allow the credentials account to read the google analytics data. This is currently free and no cost is incurred. below are the steps how to setup your analytics account, service account and obtain the credentials file.

Code

```
# %pip install google-auth-oauthlib  
# %pip install --upgrade google-api-python-client  
  
from google.oauth2.service_account import Credentials  
from googleapiclient.discovery import build  
from datetime import datetime, timedelta  
from IPython.display import SVG  
  
def get_values(response, searchKey)-> list[str]:  
    return [
```



```

        value.get('value', None)
    for row
    in response.get('rows', [])
        for key, values
        in row.items()
            for value
            in values
                if key == searchKey
    ]

def get_views(
    page_path: str,
    property_key: str,
    credentials_filename: str) -> tuple[int, list[str]]:
    SCOPES = ['https://www.googleapis.com/auth/analytics.readonly']
    credentials = Credentials.from_service_account_file(
        credentials_filename,
        scopes=SCOPES)
    analytics = build('analyticsdata', 'v1beta', credentials = credentials)
    tomorrow_date = (datetime.now() + timedelta(days=1)).strftime('%Y-%m-%d')
    body = {
        "dateRanges": [{
            "startDate": "2024-01-01",
            "endDate": tomorrow_date
        }],
        "dimensions": [{
            "name": "pagePath"
        }],
        "dimensionFilter": {
            "filter": {
                "fieldName": "pagePath",
                "stringFilter": {
                    "matchType": "CONTAINS",
                    "value": page_path
                }
            }
        },
        "metrics": [{
            "name": "screenPageViews"
        }]
    }
    response = analytics.properties().runReport(

```

```

        property=property_key,
        body=body).execute()
metricValues = get_values(response, 'metricValues')
dimensionValues = get_values(response, 'dimensionValues')

metadata = [
    f"{dimension} :: {metric}"
    for (dimension, metric)
    in zip(dimensionValues, metricValues)
]
return (
    sum([int(i or 0) for i in metricValues]),
    metadata
)

def generate_badge(label: str, value: str | int, metadata: list[str] = []) -> str:
    value_text = f"{value:," if isinstance(value, int) else str(value)

    # Calculate widths
    label_width = len(label) * 8 + 10 # Approximate width calculation
    value_width = len(value_text) * 8 + 10 # Approximate width calculation
    total_width = label_width + value_width

    # Generate metadata comments
    metadata_comments = "\n      ".join(
        [f"<!-- METADATA: {item.replace('-->', '->')} -->" for item in metadata])

    return f'''<?xml version="1.0" encoding="UTF-8"?>
<svg xmlns="http://www.w3.org/2000/svg" width="{total_width}" height="20">
  {metadata_comments}
  <linearGradient id="b" x2="0" y2="100%">
    <stop offset="0" stop-color="#bbb" stop-opacity=".1"/>
    <stop offset="1" stop-opacity=".1"/>
  </linearGradient>
  <mask id="a">
    <rect width="{total_width}" height="20" rx="3" fill="#fff"/>
  </mask>
  <g mask="url(#a)">
    <rect width="{label_width}" height="20" fill="#555"/>
    <rect x="{label_width}" width="{value_width}" height="20" fill="#4c1"/>
    <rect width="{total_width}" height="20" fill="url(#b)"/>
  </g>'''

```

```

    <g fill="#fff" text-anchor="middle" font-family="DejaVu Sans,Verdana,Geneva,sans-serif"
      <text x="{label_width/2}" y="15" fill="#010101" fill-opacity=".3">{label}</text>
      <text x="{label_width/2}" y="14">{label}</text>
      <text x="{label_width + value_width/2}" y="15" fill="#010101" fill-opacity=".3">{value}</text>
      <text x="{label_width + value_width/2}" y="14">{value_text}</text>
    </g>
  </svg>'''.strip()

```

```

def format_number(num: int|float):
    """
    Format a number similar to YouTube's style:
    - Less than 1000: show as is (100, 999)
    - Thousands: show as K (1K, 1.3K)
    - Millions: show as M (1.1M, 2.4M)

    Args:
        num (int/float): Number to format

    Returns:
        str: Formatted number string
    """
    abs_num = abs(num)
    sign = '-' if num < 0 else ''

    if abs_num < 1000:
        return f"{sign}{abs_num:d}"

    elif abs_num < 1000000:
        formatted = abs_num / 1000
        # If the decimal part is 0, don't show it
        if formatted.is_integer():
            return f"{sign}{int(formatted)}K"
        return f"{sign}{formatted:.1f}K"

    else:
        formatted = abs_num / 1000000
        if formatted.is_integer():
            return f"{sign}{int(formatted)}M"
        return f"{sign}{formatted:.1f}M"

```

Tests

```
views_count, metadata = get_views(  
    page_path = "/posts/private-domain-checker",  
    property_key = "properties/465...623",  
    credentials_filename = '../..../SERVICE_ACCOUNT_CREDENTIALS.json')  
svg_code = generate_badge(  
    label = "readers",  
    value = format_number(views_count),  
    metadata = metadata)
```

```
print(svg_code)
```

```
<?xml version="1.0" encoding="UTF-8"?>  
  <svg xmlns="http://www.w3.org/2000/svg" width="92" height="20">  
    <!-- METADATA: /posts/private-domain-checker/ :: 88 -->  
    <!-- METADATA: /posts/private-domain-checker/index.html :: 5 -->  
    <!-- METADATA: /posts/private-domain-checker :: 2 -->  
    <linearGradient id="b" x2="0" y2="100%">  
      <stop offset="0" stop-color="#bbb" stop-opacity=".1"/>  
      <stop offset="1" stop-opacity=".1"/>  
    </linearGradient>  
    <mask id="a">  
      <rect width="92" height="20" rx="3" fill="#fff"/>  
    </mask>  
    <g mask="url(#a)">  
      <rect width="66" height="20" fill="#555"/>  
      <rect x="66" width="26" height="20" fill="#4c1"/>  
      <rect width="92" height="20" fill="url(#b)"/>  
    </g>  
    <g fill="#fff" text-anchor="middle" font-family="DejaVu Sans,Verdana,Geneva,sans-serif">  
      <text x="33.0" y="15" fill="#010101" fill-opacity=".3">readers</text>  
      <text x="33.0" y="14">readers</text>  
      <text x="79.0" y="15" fill="#010101" fill-opacity=".3">95</text>  
      <text x="79.0" y="14">95</text>  
    </g>  
  </svg>
```

```
SVG(svg_code)
```

```
readers 95
```

```
views_count, metadata = get_views(  
    page_path = "/",  
    property_key = "properties/465...623",  
    credentials_filename = '../.../SERVICE_ACCOUNT_CREDENTIALS.json')  
SVG(generate_badge(  
    label = "total website views",  
    value = format_number(views_count),  
    metadata = metadata))
```

total website views 1.3K

Below is a live version of the total website views:

https://toknow.ai/pageviews?page_path=/&label=total%20website%20views

Using Cloudflare Worker

At the time of this writing, [cloudflare workers supports various languages](#)¹. However, Python and Rust are currently in beta. Python packages do not run in production, you can only deploy Python Workers that use the standard library and a [few select packages](#). That means we can't use the google python packages `google-auth-oauthlib` and `google-api-python-client`. We would have to create the authentication logic from scratch, which adds maintenance burden. Also for python, only HTTP libraries that are able to make requests asynchronously are supported. Currently, these include `aiohttp` and `httpx`. Also, python is not really a first class language in a true sense, it still uses `webassembly` and `pyodide`: <https://blog.cloudflare.com/python-workers/>.

Javascript seems to be the best candidate for a cloudflare worker. And since Typescript is also supported, its easier to use typescript for type safety and maintenance. In case the workers API changes in future, typescript will highlight the areas that are incompatible.

Implementing Typescript Worker

Despite Javascript and Typescript being the best languages to implement the worker, there is also some limitations. Not all NodeJS packages and environment are supported. As such, we'll also need to implement a google authentication from scratch. But fortunately, its easier to get a third party package compatible with cloudflare, such as <https://github.com/Schachte/cloudflare-google-auth>². This offsets alot of maintenance. <https://github.com/Schachte/>

¹Cloudflare offers first-class support for [JavaScript](#), [TypeScript](#), [Python](#), [Rust](#) and [WebAssembly](#). [WebAssembly](#) allows one to write Workers using [C](#), [C++](#), [Kotlin](#), [Go](#) and [more](#)

²<https://github.com/Schachte/cloudflare-google-auth> and https://ryan-schachte.com/blog/oauth_cloudflare_workers/

[cloudflare-google-auth](#) seems to be the package with latest maintenance, but there are other older and possibly working implementations and online discussions about authenticating and authorizing google API calls using Service Account credentials ^{3 4 5 6 7 8}

```
// https://github.com/ToKnow-ai/display-google-analytics-views-using-cloudflare-worker/blob/1
import GoogleAuth, { GoogleKey } from 'cloudflare-workers-and-google-oauth'

// JSON containing the key for the service account
// download from GCS
export interface Env {
  SERVICE_ACCOUNT_CREDENTIALS: string;
  GA_PROPERTY_ID: string;
}

const getTomorrowDate = () => {
  const tomorrow = new Date();
  tomorrow.setDate(tomorrow.getDate() + 1);
  return tomorrow.toISOString().split('T')[0];
}

// Helper function to calculate text width
const getTextWidth = (text: string): number => {
  // Approximate character widths (can be adjusted for more accuracy)
  const averageCharWidth = 8;
  return text.length * averageCharWidth + 10; // Adding padding
};

// Enhanced helper function to generate SVG badge
const generateBadge = (label: string, value: string | number, metadata: string[] = []) => {
  const valueText: string = typeof value === 'number' ? value.toLocaleString() : value;

  // Calculate widths
  const labelWidth = getTextWidth(label);
  const valueWidth = getTextWidth(valueText);
```

³Node.js helper libraries Google provide don't work with Cloudflare workers

⁴() Cloudflare Workers Google OAuth and Implementing Google OAuth to use Google API in Cloudflare Workers

⁵Web Auth Library

⁶Example: Google OAuth 2.0 for Service Accounts using CF Worker

⁷Converts Google service user OAuth2 credentials into an access token in Cloudflare-compatible JS

⁸Allow retrieving an OAuth 2.0 authentication token for interacting with Google services using the service account key

```

const totalWidth = labelWidth + valueWidth;

// Generate metadata comments
const metadataComments = metadata
  .map(item => `<!-- METADATA: ${item.replace(/-->/g, '->')} -->`)
  .join('\n');

return `<?xml version="1.0" encoding="UTF-8"?>
<svg xmlns="http://www.w3.org/2000/svg" width="${totalWidth}" height="20">
  ${metadataComments}
  <linearGradient id="b" x2="0" y2="100%">
    <stop offset="0" stop-color="#bbb" stop-opacity=".1"/>
    <stop offset="1" stop-opacity=".1"/>
  </linearGradient>
  <mask id="a">
    <rect width="${totalWidth}" height="20" rx="3" fill="#fff"/>
  </mask>
  <g mask="url(#a)">
    <rect width="${labelWidth}" height="20" fill="#555"/>
    <rect x="${labelWidth}" width="${valueWidth}" height="20" fill="#4c1"/>
    <rect width="${totalWidth}" height="20" fill="url(#b)"/>
  </g>
  <g fill="#fff" text-anchor="middle" font-family="DejaVu Sans,Verdana,Geneva,sans-serif">
    <text x="${labelWidth/2}" y="15" fill="#010101" fill-opacity=".3">${label}</text>
    <text x="${labelWidth/2}" y="14">${label}</text>
    <text x="${labelWidth + valueWidth/2}" y="15" fill="#010101" fill-opacity=".3">${value}</text>
    <text x="${labelWidth + valueWidth/2}" y="14">${valueText}</text>
  </g>
</svg>`.trim();
};

const formatNumber = (num: number) => {
  const isNegative = num < 0;
  const absNum = Math.abs(num);

  if (absNum < 1000) {
    return `${isNegative ? '-' : ''}${Math.trunc(absNum)}`;
  } else if (absNum < 1000000) {
    const formattedNum = absNum / 1000;
    return `${isNegative ? '-' : ''}${formattedNum.toFixed(formattedNum % 1 === 0 ? 0 : 1)}k`;
  } else {
    const formattedNum = absNum / 1000000;
    return `${isNegative ? '-' : ''}${formattedNum.toFixed(formattedNum % 1 === 0 ? 0 : 1)}M`;
  }
};

```

```

        return `${isNegative ? '-' : ''}${formattedNum.toFixed(formattedNum % 1 === 0 ? 0 : 1)}
    }
}

type Entries<T> = { [K in keyof T]: [K, T[K]]; }[keyof T] [];

namespace GoogleAnalyticsReport {
    type MetricType = 'TYPE_INTEGER' | 'TYPE_FLOAT' | 'TYPE_STRING';

    interface DimensionHeader {
        name: string;
    }

    interface MetricHeader {
        name: string;
        type: MetricType;
    }

    export interface DimensionValue {
        value: string;
    }

    export interface MetricValue {
        value: string;
    }

    export interface ReportRow {
        dimensionValues: DimensionValue[];
        metricValues: MetricValue[];
    }

    interface ReportMetadata {
        currencyCode: string;
        timeZone: string;
    }

    export interface Report {
        dimensionHeaders: DimensionHeader[];
        metricHeaders: MetricHeader[];
        rows: ReportRow[];
        rowCount: number;
        metadata: ReportMetadata;
    }
}

```



```

    kind: string;
  }
}

const getValues = (
  analyticsResponse: GoogleAnalyticsReport.Report,
  searchKey: keyof GoogleAnalyticsReport.ReportRow) => {
  return (analyticsResponse?.['rows'] ?? [])
    .flatMap(row => Object.entries(row) as Entries<GoogleAnalyticsReport.ReportRow>)
    .filter(([key, _]) => key === searchKey)
    .flatMap(([, values]) => values)
    .map(value => value?.['value'])
}

const IMAGE_CACHE_SECONDS = 45 * 60; // Cache for 45 minutes
const GOOGLE_CALL_CACHE_TTL_SECONDS = 45 * 60; // 45 minutes before revalidating the resource

export default {
  async fetch(
    request: Request,
    env: Env,
    ctx: ExecutionContext
  ): Promise<Response> {
    try {
      const cache = caches.default
      let response = await cache.match(request)
      if (!response) {
        if (!env.SERVICE_ACCOUNT_CREDENTIALS || !env.GA_PROPERTY_ID) {
          throw new Error("No credentials")
        }

        const scopes: string[] = [
          'https://www.googleapis.com/auth/analytics.readonly']
        const googleAuth: GoogleKey = JSON.parse(
          atob(env.SERVICE_ACCOUNT_CREDENTIALS))
        // initialize the service
        const oauth = new GoogleAuth(googleAuth, scopes)
        const token = await oauth.getGoogleAuthToken()

        if (token === undefined) {
          throw new Error("generating Google auth token failed")
        }
      }
    }
  }
}

```

```

// Only allow GET requests
if (request.method !== 'GET') {
  return new Response('Method not allowed', { status: 405 });
}

const getQuery = (queryName) => [
  ...new URL(request.url).searchParams.entries()].find(
    ([key, _]) => (key || '').toLowerCase().trim() === queryName)?.[1]
const page_path = getQuery("page_path");
if (!page_path) {
  return new Response('page_path not available', { status: 405 });
}
// Construct the request body
const requestBody = {
  "dateRanges": [{
    "startDate": "2024-01-01",
    "endDate": getTomorrowDate()
  }],
  "dimensions": [{
    "name": "pagePath"
  }],
  "dimensionFilter": {
    "filter": {
      "fieldName": "pagePath",
      "stringFilter": {
        "matchType": "CONTAINS",
        "value": page_path.toLowerCase().trim()
      }
    }
  }
},
"metrics": [{
  "name": "screenPageViews"
}]
};

// Make request to Google Analytics API
const analyticsResponse = await fetch(
  `https://analyticsdata.googleapis.com/v1beta/${env.GA_PROPERTY_ID}:runRepe
  {
    method: 'POST',
    headers: {
      'Authorization': `Bearer ${token}`,

```

```

        'Content-Type': 'application/json',
    },
    body: JSON.stringify(requestBody),
    'cf': {
        // Always cache this fetch regardless of content type
        // for a max of 45 minutes before revalidating the resource
        cacheTtl: GOOGLE_CALL_CACHE_TTL_SECONDS,
        cacheEverything: true
    },
});

if (!analyticsResponse.ok) {
    return new Response(
        JSON.stringify({ error: await analyticsResponse.text() }, null, 4),
        {
            status: 500,
            headers: {
                'Content-Type': 'application/json'
            }
        });
}

const data = await analyticsResponse.json<GoogleAnalyticsReport.Report>();
const metricValues = getValues(data, 'metricValues');
const dimensionValues = getValues(data, 'dimensionValues');

const viewsCount = metricValues
    .map(value => parseInt(value ?? '0'))
    .reduce((total, count) => total + count, 0);

// take the first 20 to avoid bloating the image. Sort desc in future..
const metadata = dimensionValues.slice(0, 20).map((value, index) =>
    `${value} :: ${metricValues?.[index]?.toLocaleString?.() ?? 'NULL'}`);

const shorten = getQuery("shorten") || false;
const label = getQuery("label") || "readers";
const badgeSVG = generateBadge(
    label,
    shorten ? formatNumber(viewsCount) : viewsCount,
    metadata);
response = new Response(
    badgeSVG,

```

```

        {
            headers:
            {
                'Content-Type': 'image/svg+xml',
                'Cache-Control': `public, max-age=${IMAGE_CACHE_SECONDS}`,
                'Access-Control-Allow-Origin': '*'
            }
        });

        // Cache API respects Cache-Control headers
        ctx.waitUntil(cache.put(request, response.clone()));
    }
    return response
} catch (error) {
    return new Response(
        JSON.stringify({
            error: error.message || 'Internal server error',
            timestamp: new Date().toISOString()
        }),
        {
            status: 500,
            headers: {
                'Content-Type': 'application/json'
            }
        });
}
},
};

```

Testing, Deployment and Troubleshooting Cloudflare Workers

- [Workers > Get started](#)
- [TypeScript is a first-class language on Cloudflare Workers](#)
- [Wrangler](#)
- [Deploy a real-time chat application](#)
- [Create a deploy button with GitHub Actions](#)
- [Explore examples of Workers](#)
- [Examples of how community developers are getting the most out of Workers](#)
- [Creating and Deploying a Cloudflare worker with Nodejs \(Typescript\)](#)

Alternatives to Cloudflare

There exist other serverless services that can offer a better service than cloudflare workers, such as Azure Functions from microsoft, Lambda Functions from Amazon Web Services and Google Cloud Functions from Google. I have extensively worked with Azure Functions for their simplicity compared to Lambda Functions or Google Cloud Functions. However, Cloudflare Workers still offer the best developer overall experience, is easier to setup, easier to use the domain path (*example.com/<any/path/can/trigger>*) to trigger your worker if your domain DNS is managed by cloudflare and doesnt require a credit card to signup and start testing & experimenting. See Table 1

Table 1: Cloudflare Workers from Cloudflare, Azure Functions from microsoft, Lambda Functions from Amazon Web Services and Google Cloud Functions from Google for a serverless architecture

Feature	Cloudflare Workers	Azure Functions	AWS Lambda Functions	Google Cloud Functions
Ease of Setup	Very easy - simple web interface and CLI tools ⁹	Moderate - requires Azure account setup and tooling ¹⁰	Moderate - requires AWS account setup and IAM configuration ¹¹	Moderate - requires GCP project setup ¹²

⁹[Workers > Get started](#)

¹⁰[Getting started with Azure Functions](#)

¹¹[Create your first Lambda function](#)

¹²[Create a Cloud Run function by using the Google Cloud CLI](#)

Feature	Cloudflare Workers	Azure Functions	AWS Lambda Functions	Google Cloud Functions
Developer Experience	Excellent - fast deployment, good documentation, simple testing ^{13 14 15 16}	Good - extensive tooling, VS Code integration ¹⁷	Good - comprehensive tooling, multiple deployment options ^{18 19}	Good - clean interface, good documentation ^{20 21 22 23}

¹³Understand how your Worker projects are performing via logs, traces, and other data sources.

¹⁴Send debugging information in an errored response to a logging service.

¹⁵Better debugging for Cloudflare Workers, now with breakpoints

¹⁶Debug via breakpoints

¹⁷Develop Azure Functions by using Visual Studio Code

¹⁸Deploying Lambda functions as .zip file archives

¹⁹Create a Lambda function using a container image

²⁰Cloud Run functions overview

²¹Create a Cloud Run function by using the Google Cloud console

²²Create a Cloud Run function by using the Google Cloud CLI

²³Create a Cloud Run function by using Cloud Code for Cloud Shell

Feature	Cloudflare Workers	Azure Functions	AWS Lambda Functions	Google Cloud Functions
Pricing	Free tier: 100,000 requests/day and 1000 requests/min per accountStandard tier: starts at \$5 USD per month ²⁴ ²⁵	Free tier: 1 million requests/- monthThen \$0.20/million requests ²⁶	Free tier: 1 million requests/- monthThen \$0.20/million requests ²⁷	Free tier: 2 million requests/monthThen \$0.40/million requests ²⁸

²⁴Users on the Workers Paid plan have access to the Standard usage model.. The Free tier offers 100,000 requests per day with no charge for duration and 10 milliseconds of CPU time per invocation. The Standard tier provides 10 million included requests per month, with additional requests costing \$0.30 per million. There's no charge or limit for duration. The Standard tier includes 30 million CPU milliseconds per month, with additional milliseconds costing \$0.02 per million. Each invocation is limited to 30 seconds of CPU time, and Cron Triggers or Queue Consumers are limited to 15 minutes per invocation. Inbound requests to your Worker. Cloudflare does not bill for [subrequests](#) you make from your Worker. Requests to static assets are free and unlimited.

²⁵Workers Paid plan is separate from any other Cloudflare plan (Free, Professional, Business) you may have. Only requests that hit a Worker will count against your limits and your bill. Since Cloudflare Workers runs before the Cloudflare cache, the caching of a request still incurs costs

²⁶Azure Functions offers a spectrum of [hosting options](#) to balance cost, control, and features.

1. [Consumption Plan](#):

- Traditional serverless model with pay-per-use billing
- Includes free grant of 1 million requests and 400,000 GB-s monthly
- Scales based on events but lacks VNet support
- Maximum 200 instances
- No cold start mitigation

2. [Flex Consumption Plan](#):

- Enhanced serverless model with more flexibility
- Supports VNet integration and always-ready instances
- Free grant of 250,000 executions and 100,000 GB-s monthly
- Faster scaling with up to 1,000 instances
- Multiple memory size options (2,048 MB and 4,096 MB)
- Includes both on-demand and always-ready billing modes

3. [Premium Plan](#):

- Enhanced performance with no cold starts
- Pre-warmed instances and VNet access
- Billing based on core seconds and memory allocation
- Requires at least one instance always running
- Scales up to 100 instances (region dependent)
- Available with 1-year or 3-year savings plans

4. [Dedicated \(App Service\) Plan](#):

Feature	Cloudflare Workers	Azure Functions	AWS Lambda Functions	Google Cloud Functions
Credit Card Required	No ²⁹	Yes ³⁰	Yes ³¹	Yes ³²

- Runs on dedicated VMs with predictable costs
- Limited to 10-30 instances depending on tier
- No auto-scaling to zero
- Best for utilizing existing App Service resources
- Includes deployment slots and Kudu console access

5. Container Options:

- Azure Container Apps (Recommended):**
 - Managed Kubernetes environment
 - Supports scale-to-zero and serverless billing
 - Includes KEDA, Dapr, and mTLS support
 - Up to 1,000 instances
 - Workload profiles for dedicated hardware/GPUs
- Azure Arc-enabled Kubernetes (Preview):**
 - Run Functions on any Kubernetes cluster
 - Managed through Azure
 - Supports both code and container deployments
- Direct Kubernetes:**
 - Open-source deployment option
 - Uses KEDA for event-driven scaling
 - Community support model
 - Requires self-management of containers

See more [Feature support comparison](#) and [Azure Functions Pricing - 2024 Guide to Azure Functions Costs & Optimization](#)

²⁷The AWS Lambda free tier includes one million free requests per month and 400,000 GB-seconds of compute time per month, usable for functions powered by both x86, and Graviton2 processors, in aggregate.. Calculate your AWS Lambda and architecture cost in a single estimate using [AWS Pricing Calculator](#)

²⁸Depending on which version of Cloud Run functions you are using, you have two [pricing options](#); [Cloud Run pricing](#) and [Cloud Run functions \(1st gen\) pricing](#)

²⁹<https://dash.cloudflare.com/sign-up/workers-and-pages>

³⁰<https://azure.microsoft.com/en-us/free/>

³¹<https://aws.amazon.com/free/>

³²<https://cloud.google.com/functions?hl=en>

Feature	Cloudflare Workers	Azure Functions	AWS Lambda Functions	Google Cloud Functions
Performance	Excellent - near zero cold start, global edge deployment ^{33 34 35 36}	Good - 100-1000ms cold start on average but also supports always ready instances ^{37 38 39 40}	Good - 100-1000ms cold start for less than 1% of invocations ^{41 42}	Good - 100-2000ms cold start ^{43 44}
Language Support	Limited - JavaScript, TypeScript, Rust, Python (beta), WebAssembly ⁴⁵	Extensive - Node.js, Python, Java, .NET, PowerShell ⁴⁶	Extensive - Node.js, Python, Java, .NET, Go, Ruby ⁴⁷	Good - Node.js, Python, Go, Java, .NET ⁴⁸

³³[Cloudflare Account plan limits, Request limits, Response limits, Worker limits\(Duration, CPU time\), and Cache API limits](#). If you are running into limits, your project may be a good fit for [Workers for Platforms](#). You can also request an adjustment to a limit by complete the [Limit Increase Request Form](#)

³⁴[Eliminating cold starts with Cloudflare Workers](#)

³⁵[Cloudflare Workers has eliminated cold starts entirely, meaning they need zero spin up time. This is the case in every location in Cloudflare’s global network. In contrast, both Lambda and Lambda@Edge functions can take over a second to respond from a cold start.](#)

³⁶[To solve this famous cold start problem, Cloudflare designed a different approach, and they claim 0ms cold starts all around the world](#)

³⁷[Azure Functions Cold start behavior](#)

³⁸[Our latest work to improve Azure Functions cold starts](#)

³⁹A typical cold start latency spans from 1 to 10 seconds. However, less lucky executions may take up to 30 seconds occasionally. [PowerShell functions are especially slow to start with values from 4 to 27 seconds](#). View detailed distributions: [Cold Start Duration per Language](#)

⁴⁰The cold start of a Function app in the Consumption plan typically ranges [between 1 and 3 seconds](#)

⁴¹[Comparison of Cold Starts in Serverless Functions across AWS, Azure, and GCP](#)

⁴²[Cold starts typically occur in under 1% of invocations. The duration of a cold start varies from under 100 ms to over 1 second](#). Provisioned Concurrency can keep your functions initialized and warm, ready to respond in double-digit milliseconds

⁴³In Cloud Run functions (1st gen), the amount of [memory granted to a function impacts the CPU allocation, which in turn can have an impact on cold start time](#). While this is also true in Cloud Run functions by default, in Cloud Run functions you have the option of configuring [CPU allocation](#) separate from memory. Latency can also be reduced by setting a minimum number of instances to avoid cold starts

⁴⁴[Comparison of Cold Starts in Serverless Functions across AWS, Azure, and GCP](#)

⁴⁵Cloudflare offers first-class support for [JavaScript, TypeScript, Python, Rust](#) and WebAssembly. WebAssembly allows one to write Workers using [C, C++, Kotlin, Go and more](#)

⁴⁶[Supported languages in Azure Functions](#)

⁴⁷[Ruby, Java, Python, Node.js, .NET](#)

⁴⁸[Cloud Run functions supports multiple language runtimes](#)

Feature	Cloudflare Workers	Azure Functions	AWS Lambda Functions	Google Cloud Functions
Integration with Other Services	Excellent with Cloudflare services ⁴⁹ Limited with external services	Excellent with Azure services ⁵⁰ Good with external services	Excellent with AWS services ⁵¹ Good with external services	Excellent with Google services ⁵² Good with external services
Scalability	Automatic, global edge network ⁵³	Automatic, regional ⁵⁴	Automatic, regional ⁵⁵	Automatic, regional ⁵⁶
Security	Built-in DDoS protection SSL/TLS by default ^{57 58}	Azure Security Center integration Key Vault integration ⁵⁹	IAM integration KMS integration ^{60 61 62}	Cloud IAM integration Secret Manager ⁶³

⁴⁹[Bindings allow your Worker to interact with resources on the Cloudflare Developer Platform.](#)

⁵⁰[Azure Functions triggers and bindings concepts](#)

⁵¹[Invoking Lambda with events from other AWS services](#)

⁵²[Access Google Cloud APIs from Cloud Run functions by using a service account to act on your behalf. The service account provides Application Default Credentials for your functions](#)

⁵³[Worker limits](#)

⁵⁴[Maximum instances are given on a per-function app \(Consumption\) or per-plan \(Premium/Dedicated\) basis, unless otherwise indicated](#)

⁵⁵[Understanding Lambda function scaling](#)

⁵⁶[Auto-scaling behavior - Cloud Run functions](#)

⁵⁷[Introducing Automatic SSL/TLS: securing and simplifying origin connectivity](#)

⁵⁸[By default, DDoS attack protection is always enabled, to customize, create a DDoS override](#)

⁵⁹[Securing Azure Functions](#)

⁶⁰[Security in AWS Lambda](#)

⁶¹[Data protection in AWS Lambda](#)

⁶²[Identity and Access Management for AWS Lambda](#)

⁶³[Secure your Cloud Run function. One way to control access to a function is to require that the requesting entity identify itself by using a credential.. You can also limit access by specifying network settings for individual functions. This allows for fine-tuned control over the network ingress and egress to and from your functions.](#)

Feature	Cloudflare Workers	Azure Functions	AWS Lambda Functions	Google Cloud Functions
Monitoring and Logging	Basic metrics and logsLimited retention ⁶⁴	Advanced Azure Monitor integration ⁶⁵	Advanced CloudWatch integration ⁶⁶	Advanced Cloud Monitoring integration ⁶⁷
Custom Domain Support	Native support with Cloudflare DNSEasy SSL setup ⁶⁸	Requires App Service domain or custom domain setup ⁶⁹	Requires API Gateway or custom domain setup ⁷⁰	Requires domain mapping configuration ⁷¹
Key Limitations	- No filesystem access- No raw TCP or UDP access- Can't use many Node.js built-in processes and modules like fs, Node crypto module etc- Other technical limits here and here	See limits here	- Maximum deployment package size of 50MB zipped (250MB unzipped)- See More limits here , here and here	See limits here

There are other options for serverless services such as [CloudFront Functions](#), [DigitalOcean Functions](#), [Oracle Cloud Functions](#), etc.

⁶⁴Understand how your Worker projects are performing via [logs, traces, and other data sources](#); [DevTools, Errors and exceptions](#), [Integrations](#), [Logs](#), [Metrics and analytics](#) and [Source maps and stack traces](#)

⁶⁵[Monitor executions in Azure Functions](#). Azure Functions offers built-in integration with [Azure Application Insights](#) to monitor functions executions

⁶⁶[Monitoring and troubleshooting Lambda functions](#). Lambda automatically monitors Lambda functions on your behalf and reports metrics through Amazon CloudWatch

⁶⁷[Monitor your Cloud Run function](#). [Google Cloud Observability](#) provides logging and monitoring tools that help you understand what is happening in your functions

⁶⁸[Custom Domains allow you to connect your Worker](#) to a domain or subdomain, without having to make changes to your DNS settings or perform any certificate management

⁶⁹[Map an existing custom DNS name to Azure App Service](#). You can use Azure DNS to manage DNS records for your domain and configure a custom DNS name for Azure App Service, see [Tutorial: Host your domain in Azure DNS](#)

⁷⁰[Custom domain name for public REST APIs in API Gateway](#). For information about custom domain names for private APIs, see [Custom domain names for private APIs in API Gateway](#)

⁷¹[Configure network settings](#). Cloud Run functions network settings enable you to control network ingress and egress to and from individual functions

Disclaimer: For information only. Accuracy or completeness not guaranteed. Illegal use prohibited. Not professional advice or solicitation. **Read more:** [/terms-of-service](#)