Barebones Broker: Migrating Snowflake's Broker to a Serverless Cloud Architecture for Censorship Circumvention

Kate Eselius
Stanford University

Lara Franciulli
Stanford University

Sophia Barnes
Stanford University

Abstract

In an era where internet censorship presents significant challenges, developing effective censorship circumvention systems is essential. This project focuses on Snowflake, a decentralized censorship circumvention system that leverages volunteer-run proxies through WebRTC technology. The central question driving this project is: Can Snowflake be improved to achieve greater scalability, higher availability, and cost efficiency? Snowflake's current implementation consists of four main components: clients, proxies, the broker and the bridge. The broker is hosted on a Tor-based server. To explore our main question, we propose transitioning the broker component of Snowflake to a *serverless* architecture using Amazon Web Services (AWS). In this work, we present a functional implementation of a serverless broker and conduct a cost analysis comparing this implementation with a server-based approach. Additionally, we identify performance bottlenecks in our implementation and outline concrete next steps to enhance the cost-effectiveness of a serverless design.

1 Introduction

Despite the ever-growing era of connectivity we live in, some governing regimes impose harsh censorship, restricting citizens' ability to freely access the internet, which has real-world consequences on education, communication, and advocacy [4]. In response, developers have created circumvention systems that combat this censorship [8] [10] [13]. These systems are essential for restoring access to information and enabling free expression, empowering individuals to overcome oppressive barriers.

In this project, we focus on Snowflake [3], which is a censorship circumvention system used and maintained by the Tor Project, Inc. This organization is a 501(c)(3) US nonprofit that receives funds from different sponsors including US federal agencies, private foundations, and individual donors [9]. Given that the organization is not self-sustaining and depends on donations, cost reduction is desirable. Additionally, according to Bocovich et al., as of March 2024, Snowflake supports around $35,000$ average concurrent users at an average total transfer rate of 2.7 Gbit/s [3]. This totals around 29 TB of circumvention traffic per day, requiring a scalable and highly available architecture.

The need for scalability, higher availability, and cost efficiency prompted us to consider migrating some of Snowflake's components to a serverless cloud architecture. More specifically, we implement a serverless Snowflake broker using AWS resources. In this paper, we describe our architecture and provide a cost analysis to compare our approach to a server-based one. Our serverless implementation is publicly available on our GitHub repository [1].

The remainder of the paper is structured as follows. Section 2 provides an overview of related work on censorship circumvention systems. Section 3 presents the benefits and drawbacks of adopting a serverless architecture. Section 4 gives an overview of the original Snowflake system, while Section 5 explains our serverless broker's architecture. We present our cost analysis comparing the serverless approach to the server-based one in Section 6, discuss future work in Section 7, and conclude in Section 8.

2 Related Work

Many various censorship circumvention systems have been developed in order to combat the rise of censorship across the world. Kon et al. created Spot Proxy to highlight the growing need for low-cost censorship circumvention systems, particularly those that can scale well even in the face of resource constraints [7]. Spot Proxy leverages low-cost cloud infrastructure to provide temporary proxies that change quickly to bypass censors. This approach significantly reduces the operational costs compared to traditional, server-hosted systems, making circumvention more accessible to users in censored regions. The authors highlight the need for lowering costs to allow circumvention systems to reach more users, especially those in economically disadvantaged areas, without sacrific-

[1] https://github.com/larafranciulli/snowflake-barebones-broker

ing the resilience or reliability of the service. Based on their work, we turned our focus toward Snowflake to see how we can change its architecture to explore alternative implementations to reduce its operational costs.

Bocovich et al. outline the Snowflake system, which consists of four main components: (1) the client, (2) the broker, (3) the snowflake proxies, and (4) the bridge [3]. In Section 4, we describe in detail the implementation of the Snowflake architecture. In our project, we focus on the rendezvous phase, which consists of two parts: the communication between client and broker, which is indirect to avoid detection (indirect rendezvous); and the communication between broker and proxy (rendezvous polling). There are three known methods to implement the rendezvous phase: domain fronting, AMP cache, and Amazon Simple Queue Service (SQS). Pu et. al [11] use Amazon SQS with a single incoming queue shared by all clients, and multiple outgoing queues, one per client. All clients can send messages to the incoming queue, but only the broker can read from it. The outgoing queues are used for the broker to send messages to a specific client once it matches it with a snowflake proxy. Their work highlights the power of using AWS in order to achieve an alternative rendezvous phase. One problem they encountered was the need to distribute public AWS credentials for clients to gain access to the Amazon SQS. Needing to distribute credentials publicly poses challenges to remain secure, especially in a censorship circumvention system. The authors discuss the additional security precautions they used such as encryption to mitigate these risks.

Various circumvention systems have different methods to connect clients to the uncensored web. Vilalong et al. emphasize the importance of rendezvous channels in censorship circumvention systems, specifically for establishing connections between users and proxies [14]. They explore how leveraging pub/sub cloud services can create resilient, censorship-resistant communication channels, allowing critical information like proxy IPs to be exchanged securely.

Frolov et al. present a system similar to Snowflake in its approach to establishing secure, censorship-resistant connections through Tor [6]. Like Snowflake, HTTPT focuses on using proxies to evade censorship, but focuses on being resistant to probing attacks. One of the main techniques is mimicking common web traffic patterns, so that the proxy traffic appears indistinguishable from regular HTTP traffic. Additionally, HTTPT uses encryption to conceal the content of communications between the user and the proxy. Finally, HTTPT responds selectively to probes, only revealing itself as a proxy under certain conditions, reducing the likelihood that a censor can detect it.

We aim to ensure that transitioning to a serverless architecture results in meaningful cost reductions for the Snowflake censorship circumvention system. Josh Barratt highlights how serverless architectures can significantly lower costs by eliminating the need for "always-on" infrastructure [2]. However, the potential cost benefits heavily depend on user traffic patterns and the specific context of the system. In his article, Barratt examines the trade-offs between a serverless architecture supported by a third-party database and a traditional server-based approach, providing cost estimates for each in the context of online gaming systems. Building on this foundation, our project seeks not only to implement a serverless broker for Snowflake but also to evaluate whether this approach would be cost-effective under the system's unique usage patterns.

3 Why Serverless Cloud?

3.1 Benefits of Serverless

Using serverless computing gives developers the ability to run code, manage data, and integrate this functionality into their applications without worrying about infrastructure management [12]. Serverless architectures are flexible and can be used to power a wide-range of applications such as mobile backends, web applications, microservices, batch processing, and data analytics [1]. For censorship circumvention in particular, infrastructure is managed by supporters out of goodwill, such as non-profit organizations like Tor. To sustain such an ecosystem, ease of maintainability and cost of operation are crucial to ensuring the longevity of such systems. By masking all infrastructure management, using a serverless architecture provides built-in high availability, scalability and fault tolerance, while avoiding developer maintenance costs for configuring and managing servers [12]. Given the number of people that rely on censorship circumvention infrastructure, any infrastructure downtime hurts users, and so highly available and fault tolerant infrastructure is critical for the success of such applications. Furthermore, by only paying for resources you use, using a serverless architecture ensures optimal utilization of resources and prevents over-provisioning. In systems like Snowflake which has varying numbers of proxies and clients, such scalability is valuable.

3.2 Drawbacks of Serverless

However, in some scenarios, applications are not well-suited for serverless architectures. First off, serverless implementations tend to rely on third-party cloud service providers which can introduce security concerns for entrusting data to these providers [5]. As noted by Pu et al., their implementation of the rendezvous phase relies on AWS, and they have observed potential real-world consequences during the deployment of their feature regarding the challenges of distributing credentials to the right parties [11]. While measures like restricting permission access and adding encryption can mitigate some risks, any dependency on third parties introduces vulnerabilities—particularly for a censorship circumvention system, which is inherently at high risk of being blocked.

In addition to security concerns associated with relying on a third-party cloud service provider, the pay-for-resources-used cost model of serverless architectures can sometimes become expensive. The cost is highly dependent on user traffic patterns. While non-consistent traffic patterns benefit significantly from paying only for individual resources used, long-running, resource-intensive applications can incur substantial and persistent charges from cloud providers [5]. Given that censorship circumvention systems often experience fluctuating usage patterns, influenced by current events, a serverless implementation could be particularly advantageous in such scenarios [3].

4 Snowflake Overview

Snowflake is a censorship circumvention system where censored clients connect to lightweight snowflake proxies via a peer-to-peer WebRTC connection, which forward the traffic to a centralized bridge [3]. Successfully deployed in the Tor browser, Snowflake represents one of the leading censorship circumvention systems with its strength coming from the fleet of more than 100,000 temporary proxies [3].

4.1 Architecture

Recall that the Snowflake system consists of four components:(1) the client, (2) the broker, (3) the snowflake proxies, and (4) the bridge [3]. The client aims to connect to the uncensored internet. The broker is a central server which matches Snowflake proxies to clients. The snowflake proxies are responsible for forwarding client traffic to the bridge. The bridge is responsible for directing traffic to its intended destination in the uncensored internet.

A Snowflake proxy connection happens in three phases. First, in the rendezvous phase, the client indicates to the broker its need for a proxy. Second, in the connection establishment phase, the client and the proxy establish a WebRTC connection between each other with the help of the broker. Third, in the data transfer phase, the proxy transfers data back and forth between the client and the bridge, which in turn will direct the client's traffic to the intended destination.

4.2 Broker

Figure 1 illustrates how the broker matches clients with proxies. Unused proxies constantly poll the broker to check for clients in need of a proxy. These proxies poll the broker every 5 seconds and timeout after 5 seconds if no client is found. A client starts a session by sending its WebRTC SDP offer to the broker. The broker matches the client to a proxy that is polling the broker and sends the client's SDP offer to the proxy. Upon receiving the client's SDP offer, the proxy forwards the broker its WebRTC SDP answer. The broker forwards the SDP answer to the client and sends an acknowledgment to the proxy.

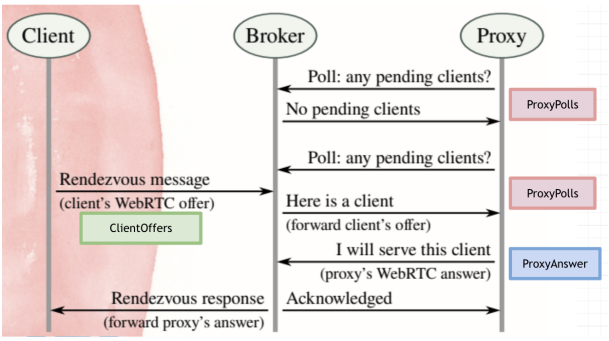

Figure 1: Broker Exchange Between Client and Proxy

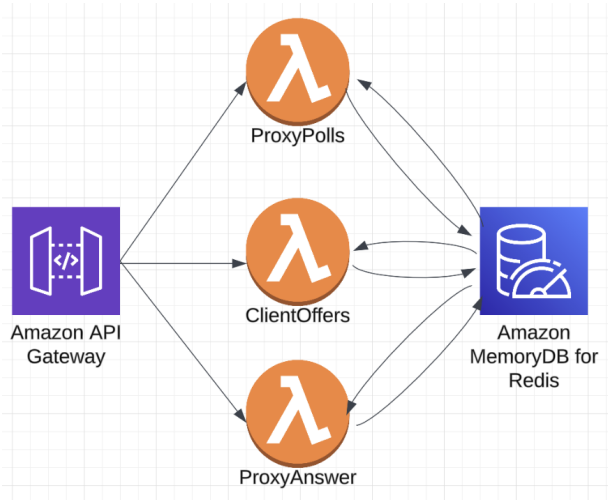

Figure 2: Serverless Broker Architecture

At this stage, the client has the information needed to initiate a WebRTC connection to the proxy.

5 Serverless Broker

Our serverless architecture is diagrammed in Figure 2. The serverless broker consists of an HTTP API with endpoints serviced by AWS Lambda functions and a Redis database. The HTTP API is hosted on AWS API Gateway. The proxy, client, and answer endpoints of the API, used by the clients and proxies to connect to the broker, are serviced by the ProxyPolls, ClientOffers, and ProxyAnswers Lambda functions respectively.

We use a Redis database hosted on Amazon's ElastiCache service to store relevant information for the client and proxy match making process. The Redis database contains four structures: (1) a table storing the proxy-client pairings, (2) a queue of proxies waiting for a client match (3) a queue of waiting client offers for each proxy, (4) a queue of waiting answers for each client. The proxy-client table contains the proxyId, proxy NAT type, clientId (empty if no paired client),

clientOffer. The queue of waiting proxies is used by the broker to match a client with a proxy. For each proxy, there is a queue of waiting clients that is populated with a client when the broker matches the client and proxy. For each client, there is a queue of waiting proxy answers that is populated when a proxy responds with the answer confirming to serve the client.

The core logic of the broker functionality is housed in the Lambda functions ProxyPolls, ClientOffers, ProxyAnswers, which we discuss in the following subsections.

5.1 ProxyPolls

ProxyPolls handles how the broker processes and responds to a proxy poll request. Proxies poll the broker at regular intervals to get matched with a client. They poll the serverless broker via the ProxyPolls Lambda function at the API's proxy endpoint. In the ProxyPolls Lambda function, the broker adds the proxyID and empty clientID to the proxy-client matching table of the Redis database. This empty clientID indicates there is currently no client match for the proxy. This clientID may get filled later if a client is matched with this proxy. We use timeouts to remove stale values from the database to ensure the database remains compact. The broker additionally adds the proxy to a Redis queue of waiting proxies. The ProxyPolls Lambda function then blocks for up to 5 seconds or until it receives a corresponding client in the queue of clients for the proxy. When the client offer arrives for this proxy, ProxyPolls will stop waiting and service the request by responding to the proxy with the client offer. If ProxyPolls reaches a timeout before a client offer arrives, the proxy receives a "no clients available" message. If no client match occurred, proxies continue polling the broker, enabling them to regularly check for new client requests to service.

5.2 ClientOffers

When a client connects to the broker through the broker's API client endpoint, the ClientOffers Lambda function services the request. The ClientOffers function first checks the Redis queue of waiting proxies. If a proxy is waiting in the queue, the broker pairs the client and proxy, adding the client's offer to the proxy's entry on the Redis database. The ClientOffers function then blocks for up to 5 seconds until it receives a corresponding proxy answer. While ClientOffers blocks, the broker adds the client offer to the proxy's queue of clients for ProxyPolls to extract the client and send the client's offer to the proxy. To unblock, the ClientOffers waits on a queue unique to the client to be populated with the matched proxy answer. If ClientOffers receives a proxy answer, the broker responds to the client's HTTP request with the proxy answer containing the WebRTC relay URL. Otherwise, it responds saying it is unable to service the request, and then the client can re-initiate the process to try again.

5.3 ProxyAnswers

Once the proxy receives a client offer, it needs to create its WebRTC relay URL to forward to the client in order to establish a peer-to-peer connection. To do so, the proxy connects to the broker's API ProxyAnswers endpoint. The WebRTC Relay URL is created by the proxy rather than the broker. The broker's responsibility is limited to simply forwarding the URL to the client. The Lambda function writes the proxy answer that includes the WebRTC relay URL to the client's queue of proxy answers, which will then notify the client through ClientOffers to accept the WebRTC relay URL. Lastly, the ProxyAnswers endpoint returns an acknowledgment response to the proxy. This way both the client and proxy have agreed upon a WebRTC URL to establish a secure connection.

6 Cost Analysis

A key motivation for exploring the feasibility of a serverless approach was the potential for cost reduction. With our functional serverless implementation of Snowflake's broker in place, we aimed to compare its monthly cost to that of a server-based implementation. Server-based solutions often incur high maintenance costs, which remain constant regardless of client activity levels. This is particularly inefficient under fluctuating client patterns, where servers must be maintained even during periods of low or no usage. In contrast, the on-demand, pay-as-you-go model of serverless computing appeared to offer a more cost-effective alternative. To evaluate this hypothesis, we conducted a cost analysis, comparing the expenses of the serverless implementation to a traditional server-based setup. The following sections detail our methodology, findings, and proposed next steps.

6.1 Methodology

To compare costs, we analyzed our serverless implementation hosted on AWS alongside an equivalent server-based implementation also hosted on AWS. While the original Snowflake paper describes hosting the broker on a server maintained by Tor, we opted not to use the original implementation for our cost comparison due to insufficient details about Tor's server expenses. Instead, we used AWS to simulate a comparable server-based setup, allowing for a direct comparison with our serverless approach.

The monthly cost of our serverless implementation was calculated by aggregating the expenses from the AWS services utilized: Lambda functions, API Gateway, Redis database, and CloudWatch Logs. Lambda functions facilitated the execution of key broker processes, including proxy polling, client offers, and proxy answers. Costs for these functions were determined based on invocation count, execution duration, and allocated memory. API Gateway enabled communication between the broker and clients, with costs linked to request volume. The

# Clients	Lambda Functions	Redis DB	API Gateway	CloudWatch Logs	Serverless Total	Server Based Cost	Server - Serverless
20000	$551356.19	$368.69	$52012.80	$0.04	$603737.72	$3664.73	$-600073.00
40000	$551480.23	$646.13	$52185.60	$0.04	$604312.00	$4254.73	$-600057.27
60000	$551604.26	$923.57	$52358.40	$0.04	$604886.28	$4844.73	$-600041.55
80000	$551728.30	$1201.01	$52531.20	$0.04	$605460.55	$5434.73	$-600025.83
100000	$551852.34	$1478.45	$52704.00	$0.04	$606034.83	$6024.73	$-600010.11

Figure 3: Cost comparison for varying client counts and 100,000 proxies for one month

# Proxies	Lambda Functions	Redis DB	API Gateway	CloudWatch Logs	Serverless Total	Server Based Cost	Server - Serverless
10000	$55495.33	$923.57	$5702.40	$0.04	$62121.34	$3378.73	$-58742.62
50000	$275988.19	$923.57	$26438.40	$0.04	$303350.20	$4554.73	$-298795.48
100000	$551604.26	$923.57	$52358.40	$0.04	$604886.28	$6024.73	$-598861.55
150000	$827220.34	$923.57	$78278.40	$0.04	$906422.35	$7494.73	$-898927.63
200000	$1102836.42	$923.57	$104198.40	$0.04	$1207958.43	$8964.72	$-1198993.70

Figure 4: Cost comparison for varying proxy counts and 60,000 clients for one month

Redis database provided storage and processing capabilities, with expenses calculated based on storage utilization and computational requirements. Lastly, CloudWatch Logs captured and stored broker-related data, with costs determined by the volume of logs. Metrics for these services were collected directly from AWS for our implementation, using rates from December 2024.

For the simulated server-based implementation, we included the following AWS components: EC2 instances, S3 storage, Application Load Balancer, network costs, and CloudWatch Logs. To ensure a conservative estimate, we simulated the server-based implementation with only two EC2 instances, recognizing that in practice, additional instances would likely be required to ensure reliability in case of failure. Standard instance and storage sizes were used to provide a lower-bound estimate. While we acknowledge that comparing directly to the current Snowflake broker implementation would be ideal, the lack of access to that information necessitated our simulated approach. Given that these AWS components could support a full implementation, we feel this represents a reasonable simulation for comparison.

6.2 Results

Figure 3 provides a breakdown of the monthly costs for each service under varying client numbers, with the number of proxies fixed at 100,000. These values are based on real traffic data reported in the original Snowflake paper [3]. Notably, our serverless broker implementation is significantly more expensive than a server-based approach—a result contrary to our initial expectations. Figure 4 further reinforces this finding, demonstrating that the serverless implementation remains more costly when simulated with 60,000 clients and varying numbers of proxies.

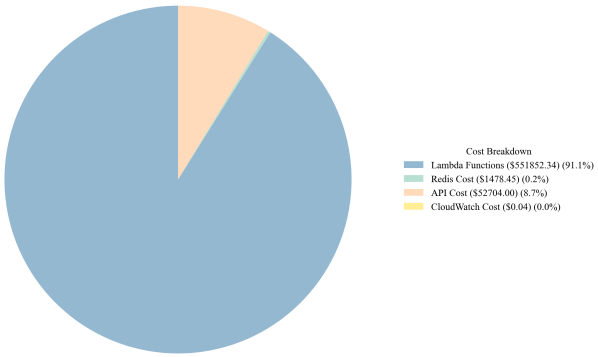

Figure 5: Cost Breakdown for 100,000 clients and proxies

Examining the cost breakdown per service in both tables reveals that Lambda functions account for the majority of the expenses, by a considerable margin. As illustrated in Figure 5, with 100,000 clients and proxies, Lambda functions constitute 91.1% of the total monthly cost. To better understand the

# Clients	# Proxies	ProxyPoll	CLientOffer	ProxyAnswer	Total Cost
20000	10000	$55123.22	$105.50	$18.54	$55247.25
40000	10000	$55123.22	$211.00	$37.08	$55371.29
60000	10000	$55123.22	$316.49	$55.62	$55495.33
80000	10000	$55123.22	$421.99	$74.16	$55619.37
100000	10000	$55123.22	$527.49	$92.70	$55743.40
20000	50000	$275616.08	$105.50	$18.54	$275740.11
40000	50000	$275616.08	$211.00	$37.08	$275864.15
60000	50000	$275616.08	$316.49	$55.62	$275988.19
80000	50000	$275616.08	$421.99	$74.16	$276112.23
100000	50000	$275616.08	$527.49	$92.70	$276236.27
20000	100000	$551232.15	$105.50	$18.54	$551356.19
40000	100000	$551232.15	$211.00	$37.08	$551480.23
60000	100000	$551232.15	$316.49	$55.62	$551604.26
80000	100000	$551232.15	$421.99	$74.16	$551728.30
100000	100000	$551232.15	$527.49	$92.70	$551852.34
20000	150000	$826848.23	$105.50	$18.54	$826972.26
40000	150000	$826848.23	$211.00	$37.08	$827096.30
60000	150000	$826848.23	$316.49	$55.62	$827220.34
80000	150000	$826848.23	$421.99	$74.16	$827344.38
100000	150000	$826848.23	$527.49	$92.70	$827468.42
20000	200000	$1102464.30	$105.50	$18.54	$1102588.34
40000	200000	$1102464.30	$211.00	$37.08	$1102712.38
60000	200000	$1102464.30	$316.49	$55.62	$1102836.42
80000	200000	$1102464.30	$421.99	$74.16	$1102960.45
100000	200000	$1102464.30	$527.49	$92.70	$1103084.49

Figure 6: Cost comparison of Lambda Functions for varying proxy and client counts for one month

source of this expense, we conducted a detailed analysis of the individual Lambda functions. Figure 6 highlights the monthly costs for each function, showing that the ProxyPolls function is the primary driver of these expenses. Specifically, under the scenario of $100,000$ clients and $100,000$ proxies, ProxyPolls accounts for 99.9% of the total Lambda cost. These findings clearly indicate that reducing the cost of the Lambda functions, particularly ProxyPolls, is critical to achieving a more cost-effective serverless implementation.

In the current implementation, which closely follows the design described in the original Snowflake paper, proxies poll the broker every 5 seconds. This polling frequency results in $51,840,000,000$ invocations of the ProxyPolls function for $100,000$ proxies over the course of a single month. Furthermore, in our implementation, once a proxy contacts the broker, it waits for up to 5 seconds to receive a client match. Consequently, the average execution time for the ProxyPolls function—particularly when client numbers are low—is approximately 5 seconds.

AWS calculates Lambda costs based on two primary factors: the number of function invocations and the duration of execution time, measured in milliseconds, with additional charges for memory allocation. In our case, the extraordinarily high number of invocations combined with a long average execution time of 5 seconds significantly drives up the cost. This combination amplifies expenses disproportionately, making ProxyPolls an exceptionally costly Lambda function within our serverless implementation. These findings underline why managing invocation rates and execution time is critical to reducing the overall cost.

6.3 Alternative ProxyPolls

To address the high costs associated with ProxyPolls, we explored alternative implementations aimed at reducing both execution time and the number of invocations, thereby improving cost efficiency. One promising approach involves replacing the current polling mechanism with an asynchronous check-in procedure, which better aligns with the architecture of Lambda functions.

In this alternative design, proxies would no longer poll the broker and wait up to 5 seconds for a match. Instead, they would be notified only when a match is available, eliminating unnecessary waiting. The broker could broadcast a request for a proxy to all available proxies and establish a match with the first successful response. To maintain a list of active proxies, the broker could implement a lightweight "heartbeat" check-in mechanism, allowing it to verify the availability of proxies with low execution times.

This asynchronous approach would retain the functionality of matching proxies with clients while significantly reducing both the execution time and invocation frequency of ProxyPolls, leading to a more cost-effective serverless implementation. We believe implementing this solution would be an immediate next step toward improving the system. However, as this approach would require modifications to both the proxy and broker codebases, it was deemed outside the scope of this project and is left as future work.

7 Future Work

We are proud to have demonstrated that it is possible to implement a serverless Snowflake broker using only AWS services. However, to fully evaluate not just the feasibility but also the cost-effectiveness of this approach in various scenarios, further work is required.

In particular, the alternative asynchronous approach outlined above should be implemented to reduce the execution time and invocations of ProxyPolls, which would likely improve the cost efficiency of the serverless broker. Additionally, further cost analysis is needed with a deployed Snowflake architecture to validate the results of our simulation. While our simulation provides a theoretical comparison, future research should focus on comparing simulated costs to actual client and proxy traffic patterns.

Key questions remain, such as how fluctuations in client traffic impact the costs of either implementation. Highly variable client traffic may align well with the on-demand nature of the serverless model, avoiding the constant costs associated with server instances in traditional setups. However, further

investigation is necessary to understand how these fluctuations affect the overall cost and performance. Future work in this area will help clarify the conditions under which a serverless implementation might be more advantageous.

8 Conclusion

In this paper, we present an alternative implementation of the censorship circumvention system Snowflake. We successfully implemented a working implementation of a serverless broker using AWS services, with a focus on AWS Lambda and AWS Redis. We conducted a cost analysis comparing our serverless implementation to a server-based approach. While our results indicate that the current serverless implementation is significantly more expensive, we identified the ProxyPolls Lambda function as the primary bottleneck driving the increased cost. Despite this, we believe that a serverless implementation has the potential to enhance the Snowflake architecture and could be cost-effective under certain traffic patterns, given the proposed change to the implementation of ProxyPolls Lambda function.

References

[1] Andrew Baird, Bryant Bost, Stefano Buliani, Vyom Nagrani, Ajay Nair, Rahul Popat, and Brajendra Singh. Aws serverless multi-tier architectures. Technical report, Amazon Web Services, 2021. https://aws.amazon.com/lambda/serverless-architectures-learn-more/.

[2] Josh Baratt. What are the limits of serverless for online gaming? *Blog*, 2021. https://serialized.net/2021/03/serverless_gaming_limits/.

[3] Cecylia Bocovich, Arlo Breault, David Fifield, Serene, and Xiaokang Wang. Snowflake: A censorship circumvention system using temporary WebRTC proxies. In *USENIX Security Symposium*, 2024. https://www.usenix.org/system/files/usenixsecurity24-bocovich.pdf.

[4] Government internet shutdowns are changing. how should citizens an democracies respond? https://carnegieendowment.org/research/2022/03/government-internet-shutdowns-are-changing-how-should-citizens-and-democracies-respond.

[5] Cloudflare. Why use serverless computing? https://www.cloudflare.com/learning/serverless/why-use-serverless/.

[6] Sergey Frolov and Eric Wustrow. Http: A probe-resistant proxy. In *USENIX Security Symposium*, 2020. https://www.usenix.org/system/files/foci20-paper-frolov.pdf.

[7] Patrick Tser Jern Kon, Sina Kamali, Jinyu Pei, Diogo Barradas, Ang Chen, Micah Sherr, and Moti Yung. Spot proxy: Rediscovering the cloud for censorship circumvention. In *USENIX Security Symposium*, 2024. https://www.usenix.org/system/files/usenixsecurity24-kon.pdf.

[8] Lantern. https://github.com/getlantern.

[9] Tor Project. Who funds tor? https://support.torproject.org/#misc_misc-3.

[10] Psiphon. https://github.com/psiphon-inc.

[11] Michael Pu, Andrew Wang, Anthony Chang, Kieran Quan, and Yi Wei Zhou. Exploring amazong simple queue service (sqs) for censorship circumvention. In *PETS symposium*, 2024. https://www.petsymposium.org/foci/2024/foci-2024-0009.pdf.

[12] Amazon Web Services. What is serverless computing? https://aws.amazon.com/what-is/serverless-computing/.

[13] Shadowsocks. https://github.com/shadowsocks.

[14] Afonso Vilalonga, Joao S. Resende, and Henrique Domingos. Looking at the clouds: Leveraging pub/sub cloud services for censorship-resistant rendezvous channels. In *PETS Symposium*, 2024. https://www.petsymposium.org/foci/2024/foci-2024-0010.pdf.