FlappyGo! - An Empirical Analysis of a Real World Case Study on The Efficacy of the Microservice Application Pattern

Ramesh Balaji, Michael Cerne, Bala Komanduri

16:198:553 Spring 2025: Design of Internet Services

Srinivas Narayana

May 10, 2025

Introduction

Recently, several new architectural patterns for internet applications have emerged. Among them, the microservice pattern promotes greater distribution by breaking down large monolithic applications into smaller, independent components. This approach is chosen at design time, with stakeholders intentionally structuring their systems around microservices to enhance reliability and performance. Most applications adopt a single architectural style—monolithic, microservice-based, or a hybrid—rather than opting to support both styles simultaneously. Very limited research exists on the effects of choosing one style or another on raw performance.

To evaluate the practical differences between monolithic and microservice architectures, we developed an internet application designed specifically to compare their performance, reliability, and other key metrics in a controlled setting. Recreating the popular mobile video game “Flappy Bird” in a client-server environment, we implemented a custom system for abstracting internal communications away into a special, swappable layer. This decision allows different parts of the application to communicate via direct function calls (in monolithic deployments) or gRPC (in microservice deployments). Nearly all other components of the application remain identical between deployment styles, minimizing external variables. We present our methodology and findings in the direct impacts on performance between architectural styles. Our repository is located at https://github.com/yuv418/cs553project and a video demo is available at https://drive.google.com/file/d/1Drgi6SjY9dhrnf5P8_sevmEN-hUAA9ZQ/view.

Methodology

There are two key components to our project: the implementation of Flappy Bird and the deployment/measurement collection system. We will detail both these components below.

Flappy Bird Implementation

Our Flappy Bird implementation is split into two parts: the backend and frontend. We will discuss the backend first.

Backend

The backend is composed of sub-components, each representing a distinct microservice. These include authentication, a game initiator, the game engine, music streaming, and a scoreboard. Initially, our project plan involved streaming frame data to the client. However, we determined this approach was too complex to implement within our timeframe. Instead, we simplified the scope by sending raw position data directly. The complexity stemmed from the need to stream frame data efficiently, which would have required sophisticated compression methods or WebRTC-based video streaming.

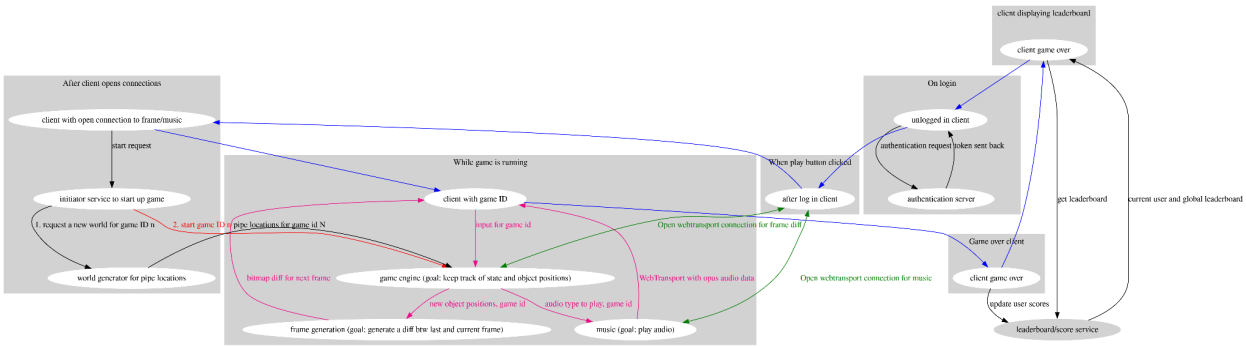


Figure 1: The initial application architecture, with frame generation still included

Although we define microservice boundaries based on these components, the monolith implementation combines them into a single binary. Each component—whether operating as a

microservice or as part of the monolith—can expose a WebTransport endpoint for client communication and support gRPC-based RPCs.

Each component provides handler functions for its RPCs. These functions accept a request context (for metadata) and the gRPC request structure and return either a corresponding response structure or an error. This consistent interface allows microservices to process requests independently while enabling the monolith to call these handlers directly, bypassing the overhead of full RPC communication.

We use generics in our gRPC server to invoke handler functions from shared boilerplate code. For each backend component, we maintain a dictionary that maps the component name to its metadata, such as the URL for microservice communication (used only in the distributed model). Another dictionary maps each RPC to its corresponding component and handler function. During setup, we enter a boilerplate code to register each endpoint route. This code handles authentication (if enabled), extracts metadata like the username, deserializes the incoming Protocol Buffer request, invokes the handler, and serializes the response back into Protocol Buffer format.

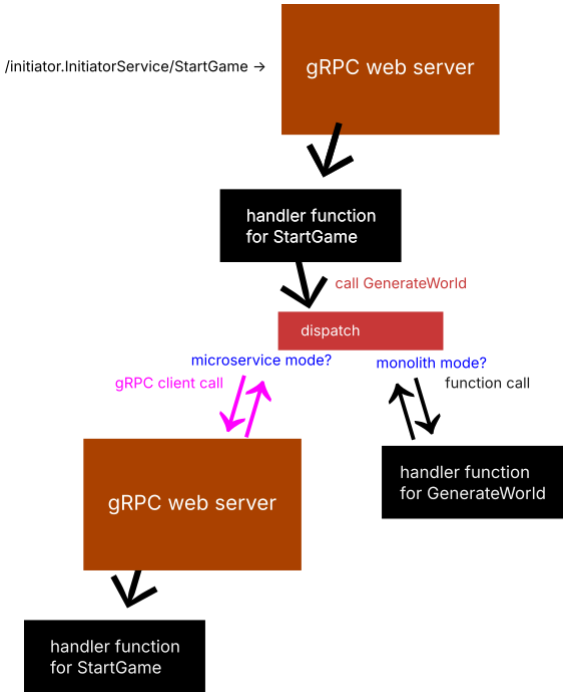


Figure 2: Example of abstraction layer in action

For WebTransport, we use similar abstractions. There simply needs to be a generic handler function to handle when data is received over WebTransport and a handler function to handle the first WebTransport connect (to save the writer to send data over WebTransport). Both these WebTransport handler functions take some additional request context. There are no RPCs to a WebTransport endpoint, so no dictionaries are required. We simply provide a function to register a WebTransport at some path with two handler functions, which configures a separate HTTP/3 server to handle WebTransport. Again, this does authentication verification, then forwards input to the input handler function, and send the transport writer to a separate registration handler function.

This abstraction layer is powerful. For each component, we have a function that registers the relevant RPCs and WebTransport endpoint (if applicable), called

`Setup<Component>Handler`. Now, we can use Go's "tags" feature to conditionally compile a specific binary for a monolith/microservice with specific component setup handlers. It is then trivial to call the various component handler setup functions for the appropriate

```
//go:build score
// +build score

package main

import abstraction "github.com/yuv418/cs553project/backend/common"

func SetupHandlers(ctx *abstraction.AbstractionServer) {
        SetupScoreHandler(ctx)
}
```

Figure 3: A microservice setup

```
//go:build monolith
// +build monolith

package main

import abstraction "github.com/yuv418/cs553project/backend/common"

func SetupHandlers(ctx *abstraction.AbstractionServer) {
        SetupAuthHandler(ctx)
        SetupWorldgenHandler(ctx)
        SetupGameEngineHandler(ctx)
        SetupInitiatorHandler(ctx)
        SetupMusicHandler(ctx)
        SetupScoreHandler(ctx)
}
```

Figure 4: The monolith setup

microservice, or register all the components for the monolith.

This abstraction layer was a key component to help us quickly test both a monolith and microservices at the same time. Updating the monolith and microservice simply involved core logic in these handler functions and rebuilding with the appropriate tags. While building the abstraction layer took significant time and effort, it provided serious gains in iteration and testing time afterwards.

The core logic of each component was easy to implement. We will briefly outline some of the functionality and some of the challenges we faced in this section.

Authentication: Supplied with the appropriate keys, we use **signed JWTs** for authentication. This eliminates the need for every component to send a request to the authentication component, lowering latency.

Initiator: This sends a request to the world generator and forwards this generated world to the game engine to tell the engine to start the game.

World Generator: This component randomly generates a fixed number of pipes (set to 100 for now) with various gap sizes and a randomly generated fixed spacing. This component was a bit complex, since we had to carefully generate these values to make sure games are playable. For instance, if the gap between two pipes is too small for the bird, the game won't work. Similarly, we also found that if the bird is physically unable to flap up to a gap or fall to a gap from its current position (e.g., the bird clears a pipe at the top of the screen and then must immediately clear a pipe at the very bottom of the screen),. To mitigate the first problem, we made sure that the pipe generator never starts a pipe in the lower ⅓ of the screen, and depending on where the pipe is initially generated, the generator will ensure that the gap is appropriately sized (e.g., a pipe gap starting towards the bottom of the screen is smaller, so the height of the gap is a larger proportion of the distance between the gap start and the bottom of the screen). To resolve the second issue of pipes with gaps at opposite ends of the screen, we make sure if a pipe gap is at the top of the screen, the next pipe gap is either at the top or the center. Similarly, if a pipe gap is at the bottom of the screen, the next pipe gap is either at the bottom or center. A pipe in the center means the next pipe can be anywhere.

Game Engine: This handles inputs to move the bird and runs a loop that constantly sends sprite positions to the client (e.g., pipe positions visible in the current frame and the bird position) at a rate of 30 FPS. The hardest part of this was taking the set of 100 generated pipe positions from the world generator and finding which pipes to tell the client to show within the viewport and where to put them (the pipes should slide smoothly as the game progresses).

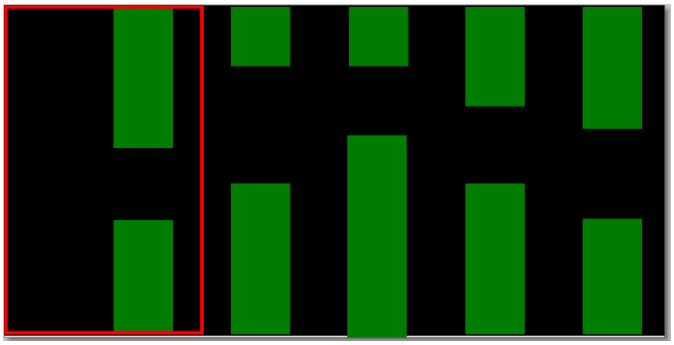


Figure 5: The initial engine state

Based on the viewport width (to calculate the max pipes to render on screen) and the left edge of the red rectangle, called the pipe X position, we can calculate what pipes to render by finding the closest pipes to the pipe X position and the difference between the X position and the closest pipe.

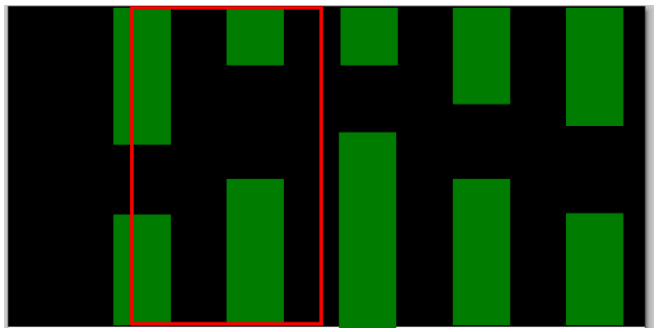


Figure 6: We will send the left pipe in a negative position, making it invisible on the canvas.

Music: Exposes an RPC to play music over the WebTransport writer. The input tells the music component what sound it should play (e.g., flapping, death sound, point sound). This component embeds Ogg Vorbis audio data as byte arrays and transmits it via WebTransport for playback.

Scoreboard: This takes a game run (with score) and updates a JSON file that serves as a database mapping users to a list of all their game runs containing score and game ID. It also exposes the list of all previous user scores, along with a global leaderboard that uses a max heap to find the highest scores efficiently.

Michael implemented the authentication logic and original boilerplate for the gRPC server. Ramesh built the abstraction and all other components.

Frontend

Our frontend/client is written in TypeScript. It uses the Connect gRPC client to interface with endpoints for each component specified by environment variables. For our monolith, the endpoints would all point to the same host and port. Endpoints for the microservices vary depending on the microservice in question. The client implements authentication and stores the JWT token sent in localStorage to avoid having to log in every time. Upon login, the client waits for the space key to be pressed, at which point it sends a gRPC request to the initiator to start the game (generating the world and forwarding it to the engine with a game ID) and starts up the WebTransports for both the music and game engine after the initiator finishes. The game engine is programmed to not actually start bird movement after. The game, which includes bird movement, does not begin until the client opens the WebTransport. This prevents the issue where the initiator starts the game but the client fails to receive frame updates for a while, resulting in the bird falling without any inputs and dying immediately.

The client uses vector graphics—namely SVGs—for rendering. The DOM renders these, and during frame updates, it modifies various CSS properties to position sprites accordingly. The music uses JavaScript's AudioContexts feature to play byte streams.

Michael created most of the client, implementing everything except the music WebTransport. Ramesh implemented the music for WebTransport and playback.

Measurement and Deployment

There are four parts to this stage of our project: instrumentation, deployment, collection, and evaluation.

Instrumentation

We first decided on what instrumentation to perform in our project.

Client:

- Authentication latency (how long does it take to log in)?
- Initiator latency (how long does it take to initiate the game)?
- Score
- Frame receive timestamps (for calculating jitter — see how far off the frames are)
- Music receive timestamps
- Input timestamps (for calculating time between input and audio received, and time between input and next frame)

It is worth elaborating slightly on the metrics we wanted to collect with the input send and frame/music receive timestamps. Essentially, it's difficult to measure any sort of round-trip time for data in WebTransport because any sort of WebTransport statistics that come from the

underlying QUIC layer are not exposed in any WebTransport library. Therefore, our goal was to determine the jitter for the frames, which should be around 1000/30 milliseconds. We can also measure the time between an input coming in and music playing, as every input will have a resulting flap sound. Note that this time will include the time it takes for the game engine to send a music request to the music component, but it is still an intriguing metric. We will explain input-to-frame time in the **collection** section.

Server: Any RPC request is timed and logged, along with the source component and destination component.

For the client, we time all measurements using JavaScript's `Performance` API. Instrumentation is generally straightforward for one-time measurements, such as initiator latency or authentication latency. For the frame and music receive timestamps, we simply use an array in memory and add a timestamp to the array every time music or a frame is received or input is being handled. This certainly adds overhead, but in our empirical testing, this overhead was not significant enough to affect the playability of the game. Our automated data collection revealed that the instrumentation's added delay did not significantly impact gameplay. When the player loses the game, a CSV file containing the direction (receive/send), the location (frame/music/input), and the timestamp is downloaded to the user's computer. The intrusive latency logging can be disabled with an environment variable.

For the server, we used Go's `time` library for measurements. We tried to minimize instrumentation overhead by using a separate thread (goroutine, which is a green thread, as Go does not support kernel threading directly). Our goal is to minimize file I/O latency during any requests that occur between components. The separate thread receives metrics over a channel and

writes them to a file. If the program terminates, the file writer flushes every write to maintain consistency in the CSV.

Bala implemented the JS frame, music, world generation, input timestamp aggregation, and CSV download. Ramesh implemented the one-shot measurements for initiator/auth latency. Ramesh implemented most of the server instrumentation, with Bala helping out.

Deployment

Michael created the deployment code for FlappyGo! Using Terraform and its AWS provider to orchestrate the FlappyGo! deployments provided a reliable foundation for managing complex infrastructure scenarios. The modular, parameterized design of our Terraform configuration allowed us to switch easily between deployment modes—monolith and microservices—and across patterns such as single instance and multi-availability zone. Multi-region support was integrated into a separate Terraform setup due to complications arising from Terraform's provider system. The deployment process featured built-in mechanisms for secure access and communication. The script handled TLS certificate provisioning via either user-supplied files or self-signed generation using local-exec provisioners, and SSH keys could be either generated or reused based on availability. Terraform resources were configured to incorporate these securely across EC2 instances, enabling encrypted channels and authenticated access without manual intervention. This setup ensured that services deployed in different configurations maintained consistent security postures.

Dynamic generation of terraform.tfvars through the deploy script simplified parameter management across environments. This approach minimized human error and enabled quick The The script transitions between test configurations by centralizing key variables, including The

deployment mode, AWS region, and instance type are important parameters. The flexibility of the script complemented Terraform's declarative model, reducing the effort needed to spin up fresh infrastructure for each experimental run greatly.

Microservices deployments leveraged Terraform to instantiate each service independently, with gRPC and WebTransport interfaces provisioned on distinct EC2 instances. Inter-service dependencies were configured post-deployment using remote-exec provisioners, which edited systemd unit files on the instances to inject the correct service URLs. While effective, this approach was sensitive to delays in instance readiness and added operational fragility. Ensuring proper sequencing and connectivity between instances required careful coordination.

Terraform provided essential structure and control over FlappyGo! deployment, supporting modular, secure, and repeatable experiments. One significant drawback encountered was Terraform's slow deploy cycle. Modest configuration changes often required full infrastructure The reapplication process led to long delays during testing and iteration. Such delays slowed development feedback loops and made experimentation more time-consuming than anticipated. While multi-region support required manual intervention and deploy times presented workflow challenges, the overall system enabled reliable testing of architectural performance tradeoffs between monoliths and microservices. We were able to easily test several distinct deployment patterns using this deployment system: monolithic (one system), microservice-based (one system), microservices deployed across multiple Amazon Web Services availability zones within one region, and micro services deployed across multiple Amazon Web Services regions (distributed globally). These findings provided us with critical data about how the application behaved in very distinct deployment scenarios.

Collection

Our goal with collection is to make it as painless as possible, à la Ousterhout. We have succeeded in this. After automating microservice and monolith deployment, we set out to automate playing the actual Flappy Bird game to avoid playing it over and over again manually when collecting data. We decided to either write a bot to play the game for us or code to record the timing of a human playing Flappy Bird and simulate these inputs. We chose the latter to reduce complexity.

The second reason we chose to simulate inputs is for *consistency*. Having consistent runs, where the inputs are sent at the same time, is beneficial. With inputs being sent at the same time across multiple runs, they are no longer variables. With the inputs standardized, we are able to measure the input-to-next-frame time. This metric, along with the input-to-music time, can be used to characterize responsiveness to some degree.

To execute the automated collection, we initially need a "standard game." We accomplished our goal by permitting the user to designate a fixed world generator seed, ensuring the identical world is produced with each execution. We conducted experiments with various seeds to identify two difficulty levels: one designed for brief play, characterized as easy with fewer "tight" inputs, and another featuring tighter inputs, classified as a harder level.

With these games fixed, we wrote a keylogger program to dump the timestamp in CSV format to standard output every time the space (flap) key is pressed. This program targets a Linux desktop and pulls data from `libinput`, the input system for Linux. This program was originally written in Python but had performance issues and inaccuracies, so we rewrote it in Rust.

To record a game, we open the game in the web browser, log in, then start the keylogger and redirect its standard output to a file. We then play the game as normal, and when finished, we

Terminate the keylogger. The CSV traces are stored in `client-automation/input_seeds`.

To simulate a game, we use a Python script that invokes Selenium with ChromeDriver, a browser automation framework. Selenium clears localStorage and logs in (to test auth latency), starts the game, and sends inputs to the game at the correct offsets. Between the first space pressed and the actual game loading, there is a delay caused by the initiator. The input traces do not account for this.

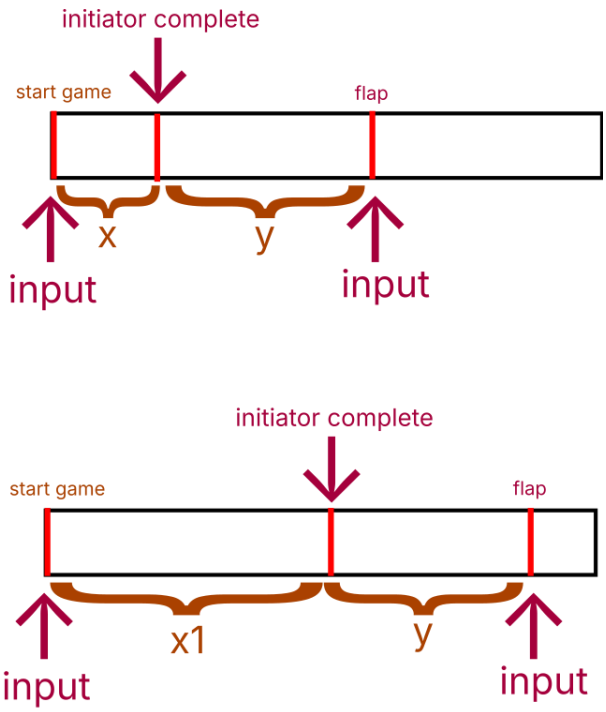


Figure 7: Initiator delay

In our traces, we only have x + y. Call this quantity z. We want to wait for the initiator to be complete, then wait for y milliseconds, as the initiator may wildly vary in its completion time—depicted in the second regime of the figure. This issue squarely depends on deployment.

To resolve this, we experimentally determined x, as we had not saved our initiator latency when taking the input trace. Then, the input simulator waits for the initiator to complete (we do this in Selenium repeatedly checked a global variable in the client's JavaScript and waited for the value of z minus x (where x is experimentally determined) after the initiator was complete.

The input simulator downloads the latency values in the client to a specified directory after the game is over, which is detected by searching for a change in DOM elements after each space is sent. This check did not seem to affect the simulation's accuracy.

With a completed deployment on Terraform, Terraform can expose JSON configuration information about the various microservice endpoints. Using this configuration, we wrote a shell script. This script first restarts all services to clear out server logs. Then, it SSHes into the world generator and fixes a seed. The script then runs the Selenium input simulator five times and saves statistics. In case the input simulator fails for some reason (such as Chrome crashing) or the first input is not recognized, the script retries this run until it is able to get a valid result. The script does this by checking if the score outputted by the input simulator is greater than zero. Automation has greatly benefited from this error checking, particularly in more distributed microservices where the game starts with unreliable inputs. This script is run with the two aforementioned seeds. At the end, we fetch all remote server logs, and everything is saved to a common directory for data processing.

In the end, we have two commands: the deploy.sh script to deploy FlappyGo on AWS with Terraform and this script. These two scripts used in succession automatically collect all logs and simulate tests, which is very convenient for testing.

Michael worked on the remote log fetching script. Ramesh worked on automating game playback and the data collection script.

Evaluation

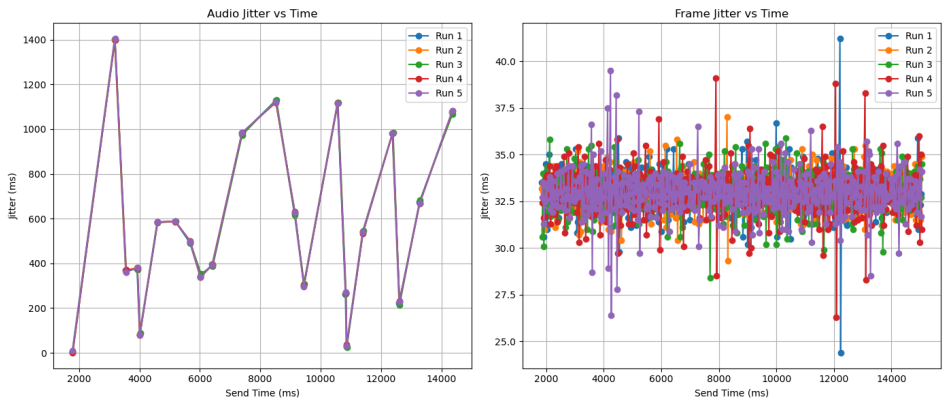
Bala generated all the graphs, averages, and visualization code. This was done in Matplotlib.

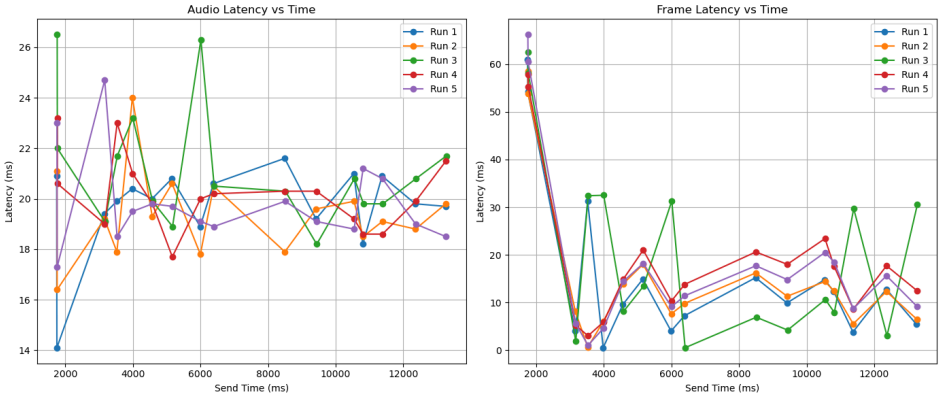
We generated various plots and split our plots by the “hard” and “easy” seeds to be able to compare these two games separately. We combine all five runs across each seed in one plot. Data local to a single run (like differences in frame receive times) are computed for just that run and combined later. **To save space, we won't show the hard-level graphs, but we will discuss some intriguing results.**

Server logs are an aggregate of all 10 runs. Failed runs are included in these server logs.

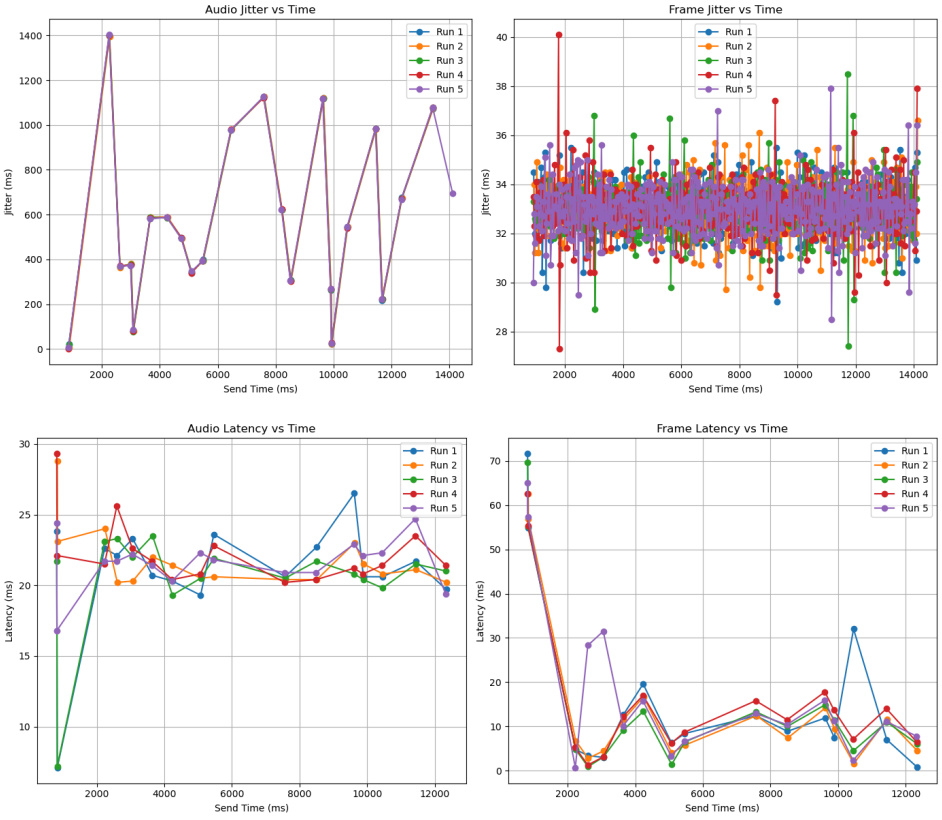
For the client logs, the top two graphs represent the time difference between successive audio or frame arrivals. Frame position data arrivals are expected every 33.3 milliseconds (the engine runs at 30 FPS), while audio data arrives after an input, after score increases, or after the game is over. The bottom two graphs indicate input-to-audio time and input-to-frame time, respectively.

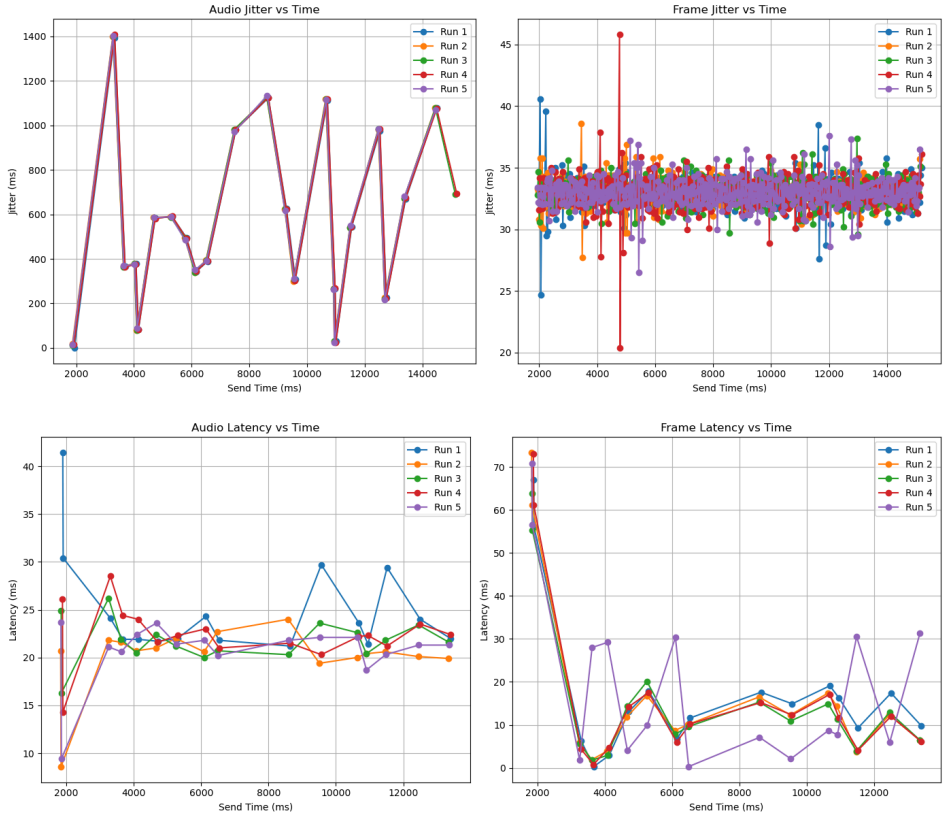
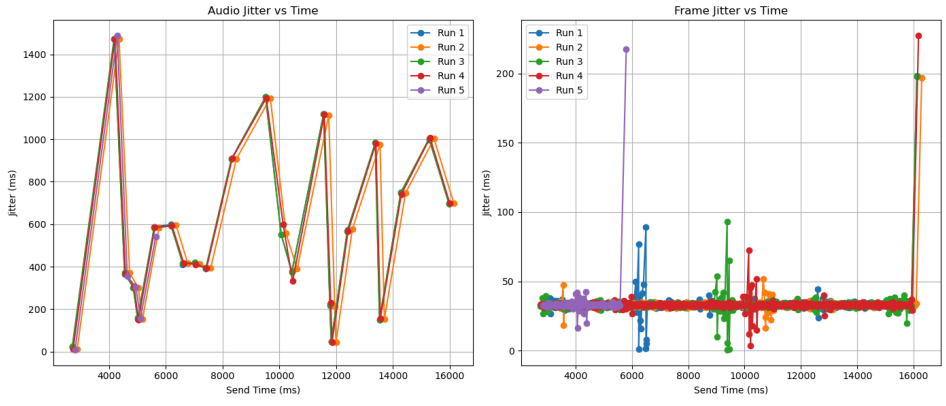
Easy Game

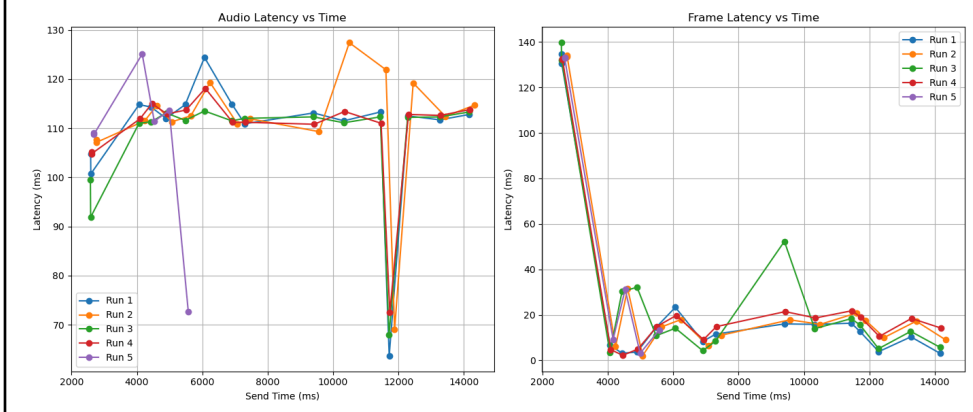
Monolith	



Microservices Single AZ



Microservices Multi-AZ	
Microservices Multi-Region	



First of all, audio is input-dependent and level-dependent because it comes in with every input and every time the score increases. This is why we have split our graphs by each world, which has the same seed and automated input.

Monolith: Frame jitter is nearly exactly centered at 33.3 ms (at 30 FPS, anticipate data arrival every 33.3 ms). Although there are minor variances, the majority of frames remain within 10 milliseconds of 33.3 ms. Regarding input-to-audio and input-to-frame, we anticipate that the subsequent frame from an input will be no more than 33.3 ms away (assuming the input is activated immediately thereafter). Most inputs appear to be within 30 milliseconds of the actual answer. Nonetheless, the initial inputs require a considerable amount of time to elicit a response. The delay is likely due to our initial input occurring prior to the commencement of the game; this interval encompasses the initiator and WebTransport connections. This will apply to all subsequent deployments.

Microservices Single-AZ: Since the services are split but nearby, we should expect similar results, but slightly worse. All services were deployed within a singular Amazon Web Services availability zone, comprising one or more closely situated data centers. Requests have to go through networking equipment in the data center, which may slightly increase latency. We do not

anticipate that the latency will not increase significantly because data center-grade networks are typically highly performant. Indeed, this appears to be the case, with slightly looser ranges in frame jitter and more frames being off the 33.3 ms “clock rate.” It’s worth noting that the input-to-audio time is now squarely above 20 ms, while in the monolith many more points were below the 20 ms mark. The data from Run 5 of the input-to-frame graph appeared to be more volatile. So we see 1-2 ms more latency and a little more volatility.

Microservices Multi-AZ:

Multi-Region: This is clearly the worst performance. We can see from the graphs that while the frame jitter is still acceptable, the arrival time of the last frame will clearly be much higher, from the outliers that take over 200 milliseconds to arrive! Compared to the Multi-AZ deployment, the time delay between frames can easily exceed 50 milliseconds. From an empirical perspective, the multi-region game feels the worst to play. This frame jitter graph shows that it is mostly smooth but has stutters. We also observed that audio comes in very late after an input. This observation is confirmed by the input-to-audio graph. Every input seems to take upwards of 100 ms to get an output—that’s significantly higher than the worst latency of multi-AZ, which is a little less than 30 ms. In fact, no audio comes in faster than 70 ms. This slowdown is caused by the latency of sending an RPC from the game engine to a music service in a different region (EU-West to EU-Central). It’s also fascinating to note how the initial space-press in input-to-frame is much higher because of the initiator performing RPCs across multiple regions. Again, we see latency spikes in the input-to-frame.

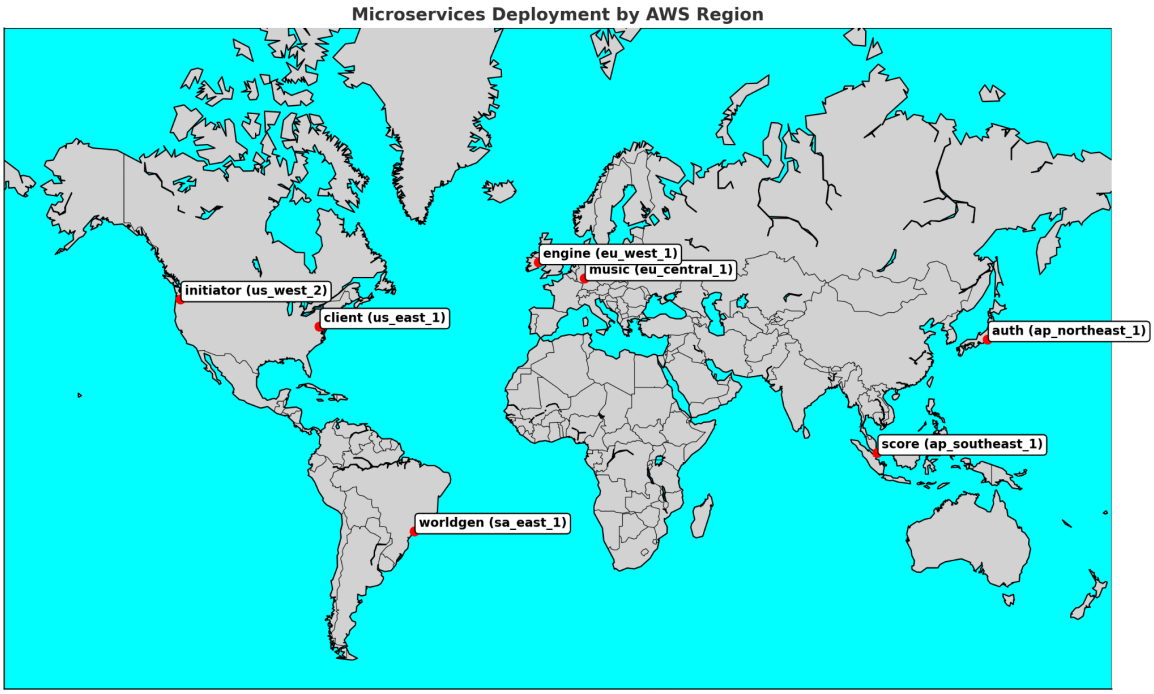


Figure 8: The locations of the various microservices in our multi-region testing

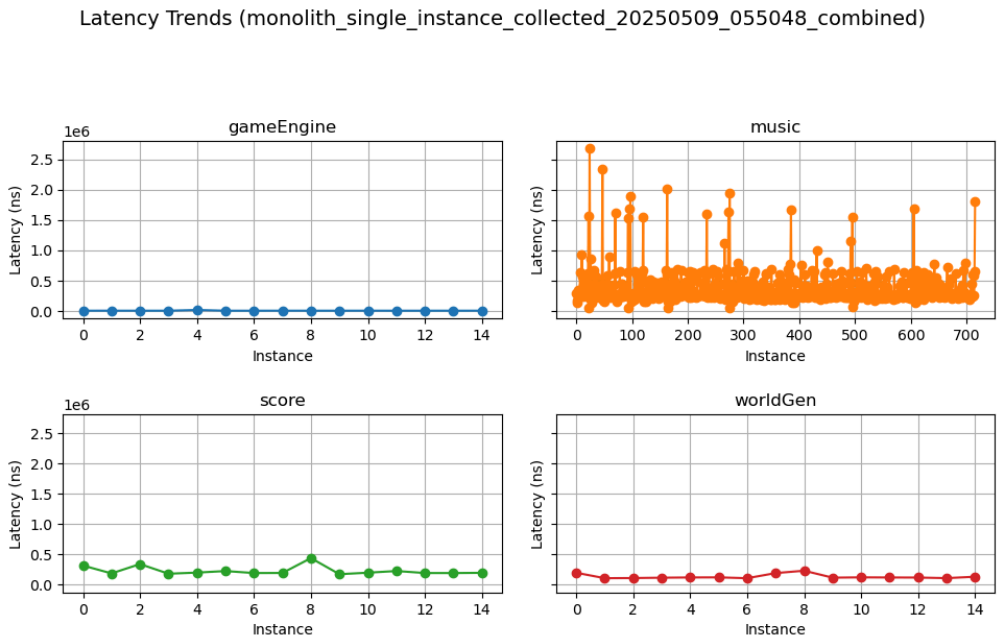
Audio jitter: It's fascinating to see how, as the services get further and further apart, the "audio jitter" (time between audio being received) gets more variable. Looking at the multi-region instance, this seems quite evident, where each run varies more and more compared to previous runs. This effect is also visible, but to a lesser extent, in the Multi-AZ runs. Single-AZ and Monolith have less of this. Comparing individual data points, the first three runs seem to exhibit similar latencies, while multi-region performs noticeably worse in every data point. The second audio jitter point is well above 1400 ms, while in all previous deployments they are around 1400 ms. This highlights how multi-region performs visibly more poorly than other deployments.

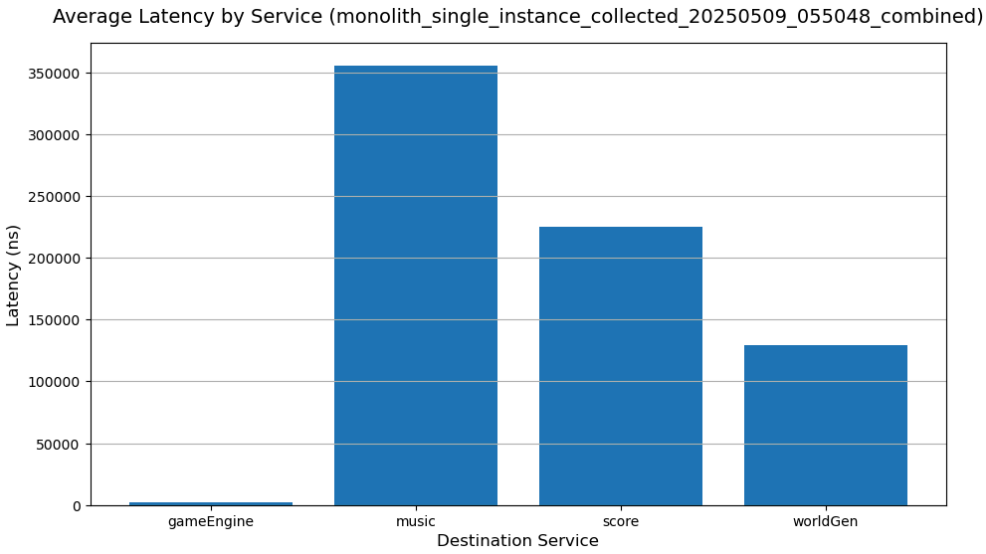
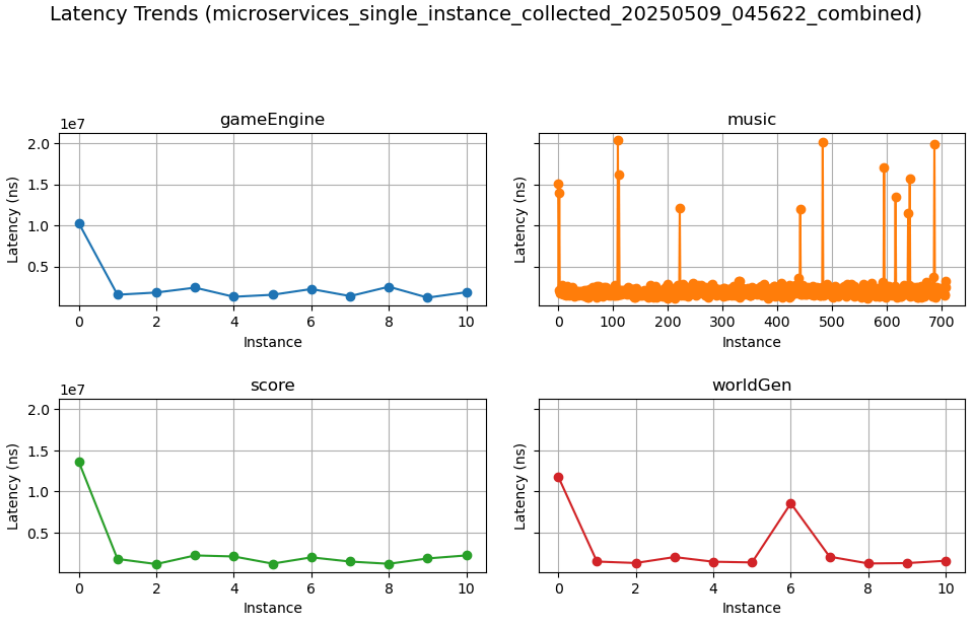
We anticipated that as input latency increases, the bird's inputs might not register promptly in the "difficult game" and the "easy game," potentially leading to a game over before reaching the expected score. For the easy game, the expected score was 7, and for the difficult game the expected score was 31. All deployment configurations reached the expected score *except* for the

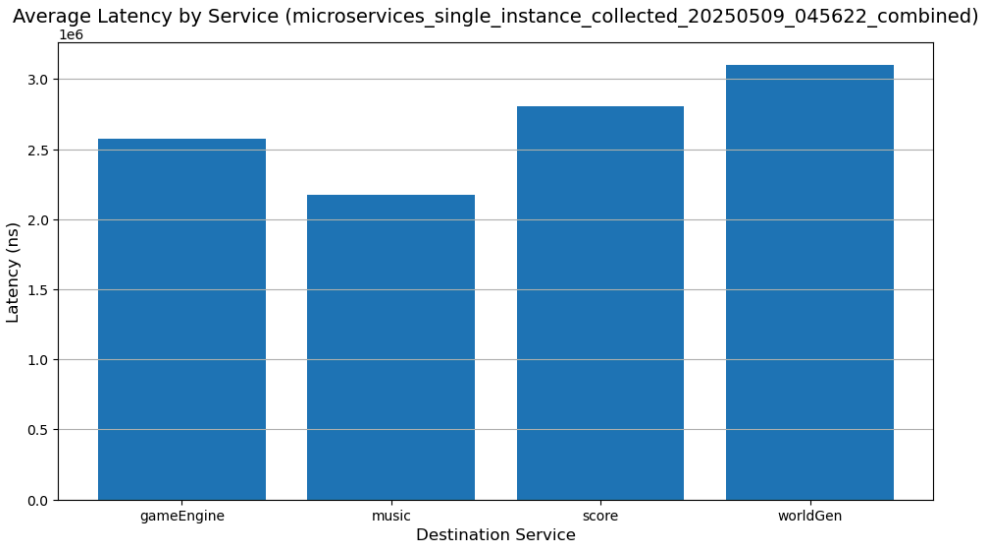
multi-region deployment, which failed once and received a score of 1; additionally, in the difficult runs, this same deployment failed once more and achieved a score of 22. This is significant because scores of 15 and 22 in the difficult game were considered "tight" inputs, which supports our hypothesis. We might need an even worse latency between the client and game server to have systematic failures here.

If there is any takeaway from these metrics, it is that latency may not significantly get worse from one point to another point, but tail latency gets worse and random spikes occur. Additionally, interactions that require multiple hops across different regions (e.g., music RPC calls) worsen as microservices move further apart.

Server Graphs

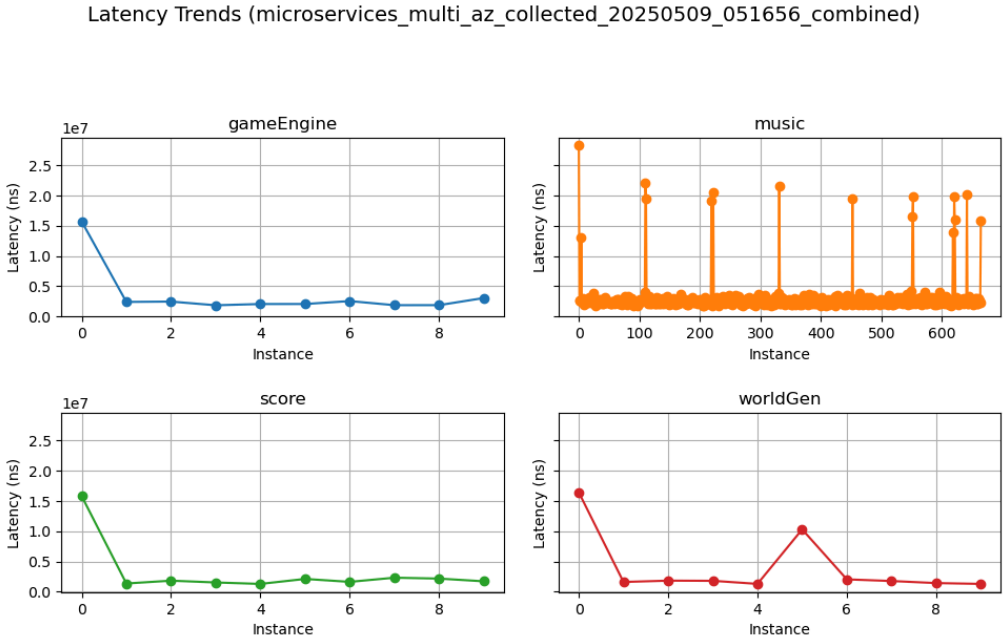
Monolith	

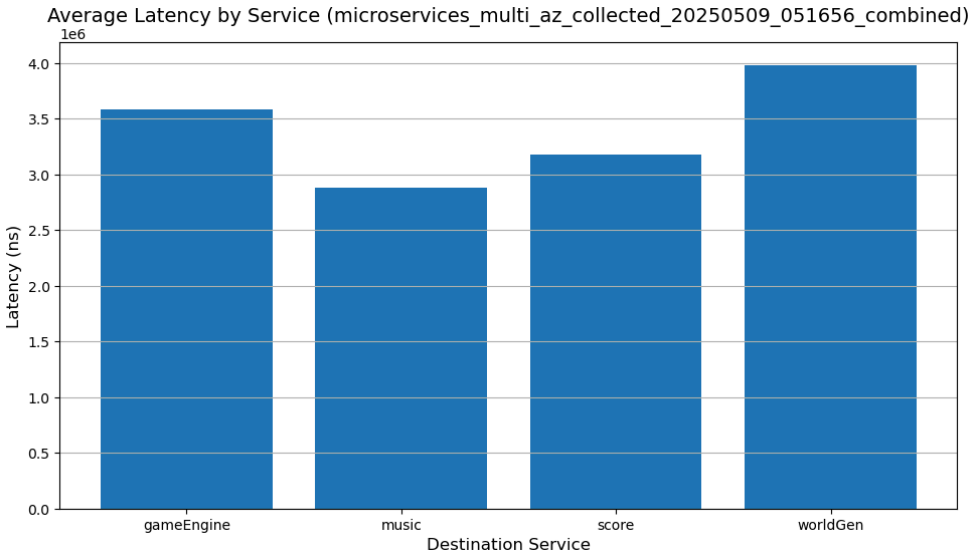
	 Average Latency by Service (monolith_single_instance_collected_20250509_055048_combined) Latency (ns) 350000 300000 250000 200000 150000 100000 50000 0 gameEngine music score worldGen Destination Service
Single AZ	 Latency Trends (microservices_single_instance_collected_20250509_045622_combined) gameEngine music score worldGen 1e7 Latency (ns) Instance



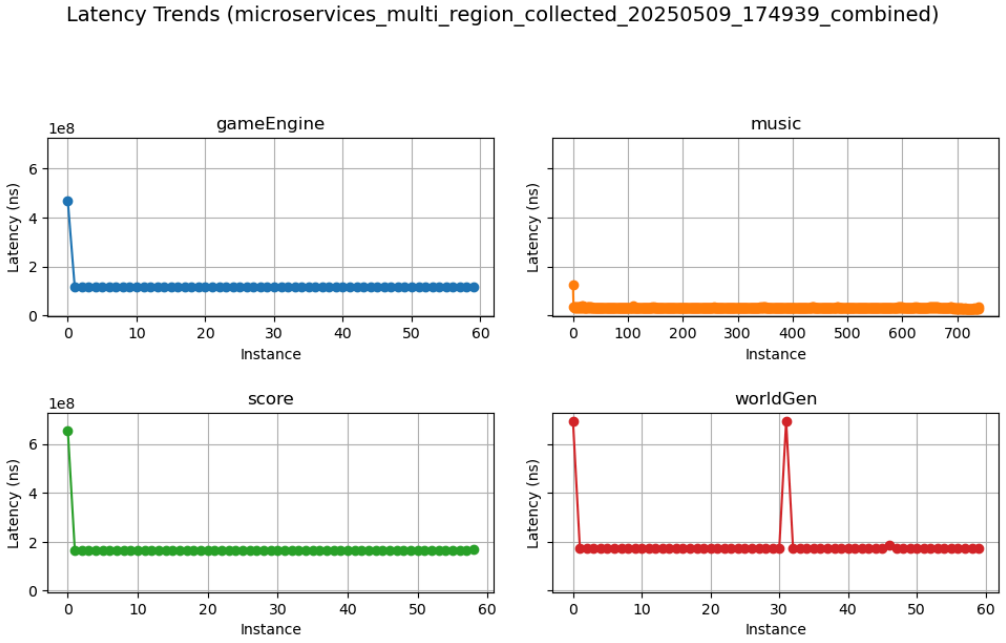
Note: “single_instance” in this context refers to deployment within a single Amazon Web Services availability zone (AZ).

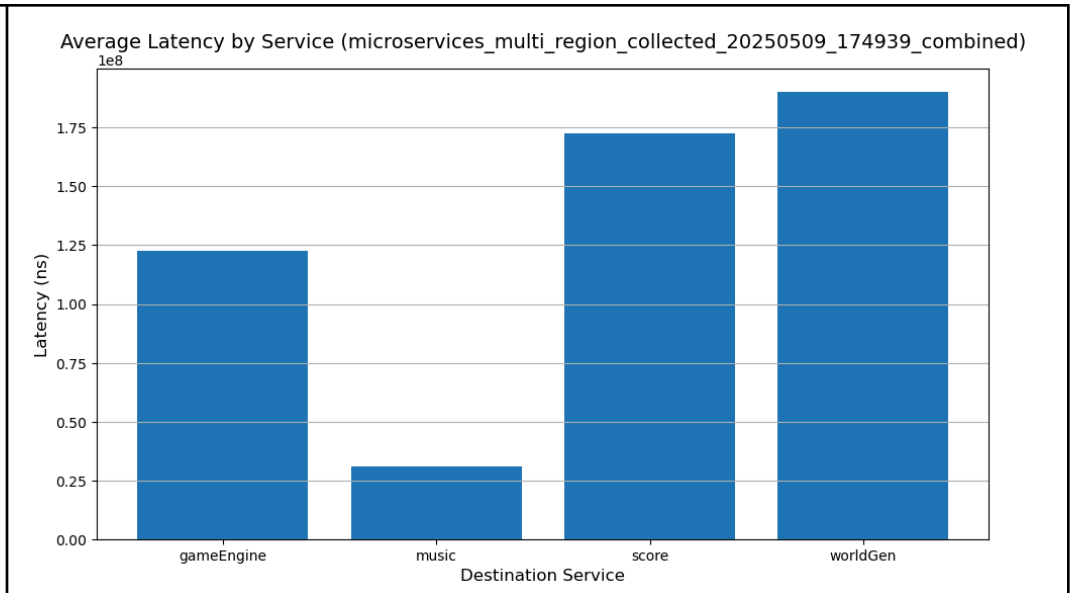
Multi AZ





Multi Region





First of all, we must pay close attention to the y-axes. It is easier to interpret the graphs by understanding the scale.

Monolith: 350000 ns = 0.35 ms (hundredths of milliseconds); Single AZ: 10^6 ns is 1 ms, in milliseconds. Multi AZ: same scale; multi-region: hundreds of milliseconds. This jump in latency is on a scale of 10x, as the time scale multiplies by ten when going from a monolith to microservices deployed in multiple machines in one region (milliseconds) and 100x from one region to microservices deployed in multiple regions (hundreds of ms). It clearly shows the decomposition of a monolith to microservice architecture is not always optimal.

Initially, the world generator experiences spikes due to the seed changing every 5 runs. The world generator creates a new random number generator and sets a seed exactly once when it first generates a world; hence this latency. This is also the reason there is a latency spike at the beginning of the world generator. However, every service has a latency spike at the beginning. This is potentially because gRPC clients seem to open persistent connections, so it is possible the

The first gRPC call takes additional time because it must actually establish a connection with the microservice first, which it reuses later on.

Another intriguing question is the regular latency spikes in the music service. This likely occurs because the score sound is twice the size of the flap sound, and the Protobuf serialization process copies the audio data, which likely increases latency.'

We shall now clarify certain irregularities in the average latencies. The multi-region code seems to be significantly influenced by regional closeness. The initiator invokes the game engine and environment generator, which subsequently calls both the music service and the score service. At present, all of these RPCs are The RPC linking the game engine to the music service is the sole one positioned near the others, while all other RPCs are distanced from one another. This is why the music RPC has a much lower average latency than other RPCs. For the in-the-monolith architecture, we observe that the game engine RPCs (from initiator to game) exhibit significantly lower latency compared to other types of Remote Procedure Calls (RPCs). The reason for this is that the StartGame RPC requires minimal computational resources. It locks a dictionary, does a few arithmetic operations in O(1) time, inserts an item into the dictionary, and finishes. This figure is in comparison to the score, music, and world generator, which perform file I/O, network I/O, and repeated random number generations, respectively. Therefore, the average engine latency for other services is likely a representation of round-trip time. Since all non-monolith services are running across different machines, we contend that the game engine's average latency being higher than other RPCs indicates that the machine is physically further away, even if in the same AZ or region.

Other than this, many of the results we see are straightforward. The time scale increases as services get progressively further apart and latency gets progressively worse.

Future Work

We took many more measurements than this (e.g., client-side RPC latencies to auth and initiator), but didn't get to plot these graphs. You can see how these measurements are expressed in our sample video. We could also try to correlate statistics by game ID to get per-game data. Additionally, we may want to experiment with performance results related to redundancy and load balancing (using an orchestration system like Kubernetes) or when multiple users are playing simultaneously. It would also be intriguing to add additional microservices or induce a chain of RPCs to see how this affects latency. We would like to explore the performance of network schedulers in mitigating the latencies by scheduling multiple flows (frame in this case)

References

Large Language Models were used occasionally for troubleshooting. Transcripts of these chat conversations were included directly in the codebase where their input was considered.

https://developer.hashicorp.com/terraform/docs

https://vite.dev/guide/

https://grpc.io/docs/

https://protobuf.dev/overview/

https://docs.aws.amazon.com/ec2/

https://docs.aws.amazon.com/vpc/

https://jqlang.org/manual/

https://matplotlib.org/stable/index.html

https://developer.mozilla.org/en-US/docs/Web/API/WebTransport

https://stackoverflow.com/questions/26388405/chrome-disable-ssl-checking-for-sites

https://stackoverflow.com/questions/47274254/how-do-i-run-terraform-init-from-a-different-folder

https://stackoverflow.com/questions/47130406/extending-global-types-e-g-window-inside-a-typescript-module

https://stackoverflow.com/questions/56431721/why-go-get-u-takes-a-very-long-time-inside-a-module-directory-but-complete-qui

https://stackoverflow.com/questions/57556909/ssl-alert-number-46-alert-certificate-unknown-how-to-ignore-this-exceptions

https://stackoverflow.com/questions/57556909/ssl-alert-number-46-alert-certificate-unknown-how-to-ignore-this-exceptions

Backend

https://gobyexample.com/channels

https://pkg.go.dev/google.golang.org/protobuf/encoding/protojson

https://pkg.go.dev/os#File.Write

https://pkg.go.dev/encoding/json

https://bitfieldconsulting.com/posts/map-iteration

https://stackoverflow.com/questions/21950244/is-there-a-way-to-iterate-over-a-range-of-integers

https://stackoverflow.com/questions/57278822/sending-grpc-communications-over-a-specific-port

https://gist.github.com/marzocchi/c4d3e2254853c5ff02b420044e796aea

https://sahansera.dev/building-grpc-client-go/

https://pkg.go.dev/google.golang.org/grpc#ClientConn.Invoke

https://www.freecodecamp.org/news/new-vs-make-functions-in-go/

https://chatgpt.com/share/680de978-f87c-8012-bd76-a8a6ae618438

https://pkg.go.dev/github.com/golang-jwt/jwt/v5#example-Parse-Hmac

https://github.com/dgrijalva/jwt-go/blob/master/hmac_example_test.go

https://pkg.go.dev/connectrpc.com/connect#NewUnaryHandler

https://gist.github.com/filewalkwithme/0199060b2cb5bbc478c5

https://stackoverflow.com/questions/69573113/how-can-i-instantiate-a-non-nil-pointer-of-type-argument-with-generic-go/69575720#69575720

https://pkg.go.dev/bufio#Writer.Flush

https://stackoverflow.com/questions/15407719/in-gos-http-package-how-do-i-get-the-query-string-on-a-post-request

https://pkg.go.dev/io#ByteScanner

https://pkg.go.dev/net/http#Header

https://www.reddit.com/r/golang/comments/cgbkel/why_are_headers_mapstringstring/

https://pkg.go.dev/strings

https://go.dev/doc/tutorial/handle-errors

https://stackoverflow.com/questions/71114401/grpc-how-to-pass-value-from-interceptor-to-service-function

https://chatgpt.com/share/6810f51b-79b8-8012-8ec9-4d526dd1c434

https://connectrpc.com/docs/go/errors/

https://gobyexample.com/timers

https://stackoverflow.com/questions/16466320/is-there-a-way-to-do-repetitive-tasks-at-intervals

https://stackoverflow.com/questions/29721449/how-can-i-print-to-stderr-in-go-without-using-log

https://gobyexample.com/timers

https://stackoverflow.com/questions/16466320/is-there-a-way-to-do-repetitive-tasks-at-intervals

https://pkg.go.dev/math/rand

https://stackoverflow.com/questions/3574716/date-and-time-type-for-use-with-protobuf

http://api.acme.com or https://acme.com/grpc).

Client Automation

https://www.geeksforgeeks.org/how-do-i-pass-options-to-the-selenium-chrome-driver-using-python/

https://stackoverflow.com/questions/5137497/find-the-current-directory-and-files-directory

https://stackoverflow.com/questions/5664808/difference-between-webdriver-get-and-webdriver-navigate

https://selenium-python.readthedocs.io/locating-elements.html

https://selenium-python.readthedocs.io/navigating.html

https://stackoverflow.com/questions/32098110/selenium-webdriver-java-need-to-send-space-key press-to-the-website-as-whol

https://stackoverflow.com/questions/46361494/how-to-get-the-localstorage-with-python-and-selenium-webdriver

https://stackoverflow.com/questions/26566799/wait-until-page-is-loaded-with-selenium-webdriver-for-python

https://stackoverflow.com/questions/14257373/how-to-skip-the-headers-when-processing-a-csv-file-using-python

https://smithay.github.io/smithay/smithay/backend/libinput/struct.LibinputInputBackend.html

https://stackoverflow.com/questions/62501219/how-to-send-keys-to-a-canvas-element-for-longer-duration

https://stackoverflow.com/questions/1133857/how-accurate-is-pythons-time-sleep

https://www.geeksforgeeks.org/python-strftime-function/

https://stackoverflow.com/questions/71716460/how-to-change-download-directory-location-path-in-selenium-using-chrome

https://stackoverflow.com/questions/54571696/how-to-hard-refresh-using-selenium/54571878#54571878

https://stackoverflow.com/questions/59130200/selenium-wait-until-element-is-present-visible-and-interactable

https://stackoverflow.com/questions/62501219/how-to-send-keys-to-a-canvas-element-for-longer-duration

https://github.com/Smithay/input.rs

https://stackoverflow.com/questions/4412238/what-is-the-cleanest-way-to-ssh-and-run-multiple-commands-in-bash

https://stackoverflow.com/questions/49110/how-do-i-write-a-for-loop-in-bash

https://serverfault.com/questions/7503/how-to-determine-if-a-bash-variable-is-empty

Paper

https://pkg.go.dev/google.golang.org/grpc/keepalive#ServerParameters