NicePiano

Cloud Computing 2020/2021

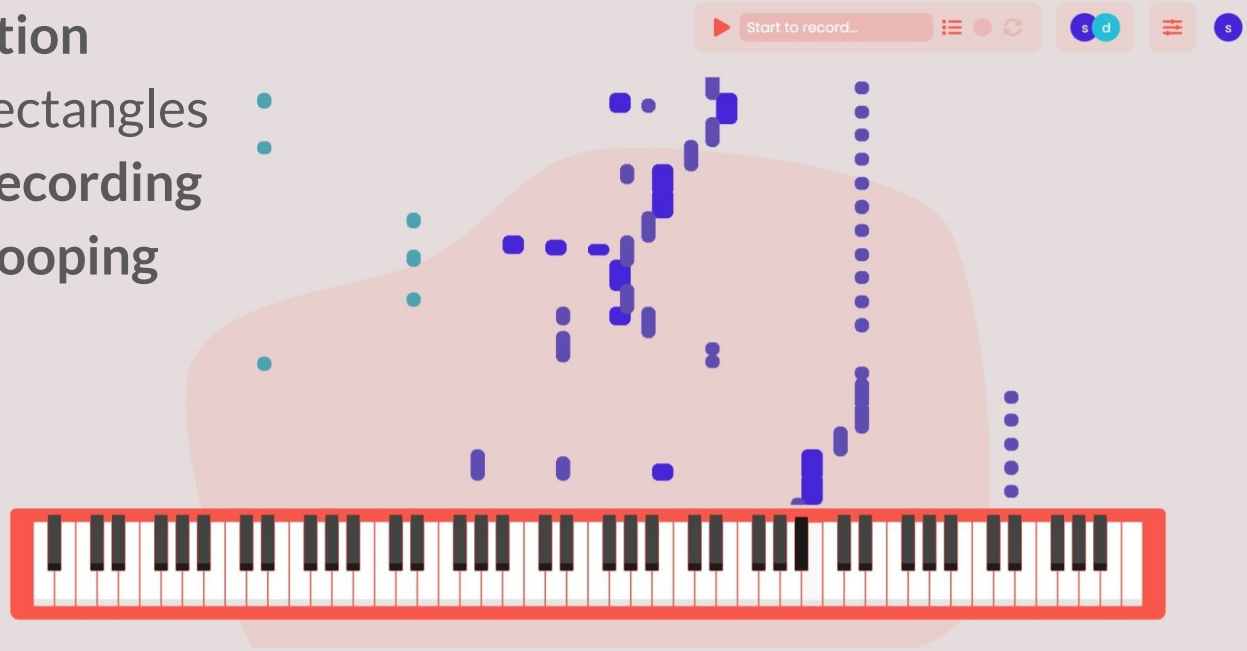
Matteo Orsini 1795119
Fabrizio Rossi 1815023

Main Features

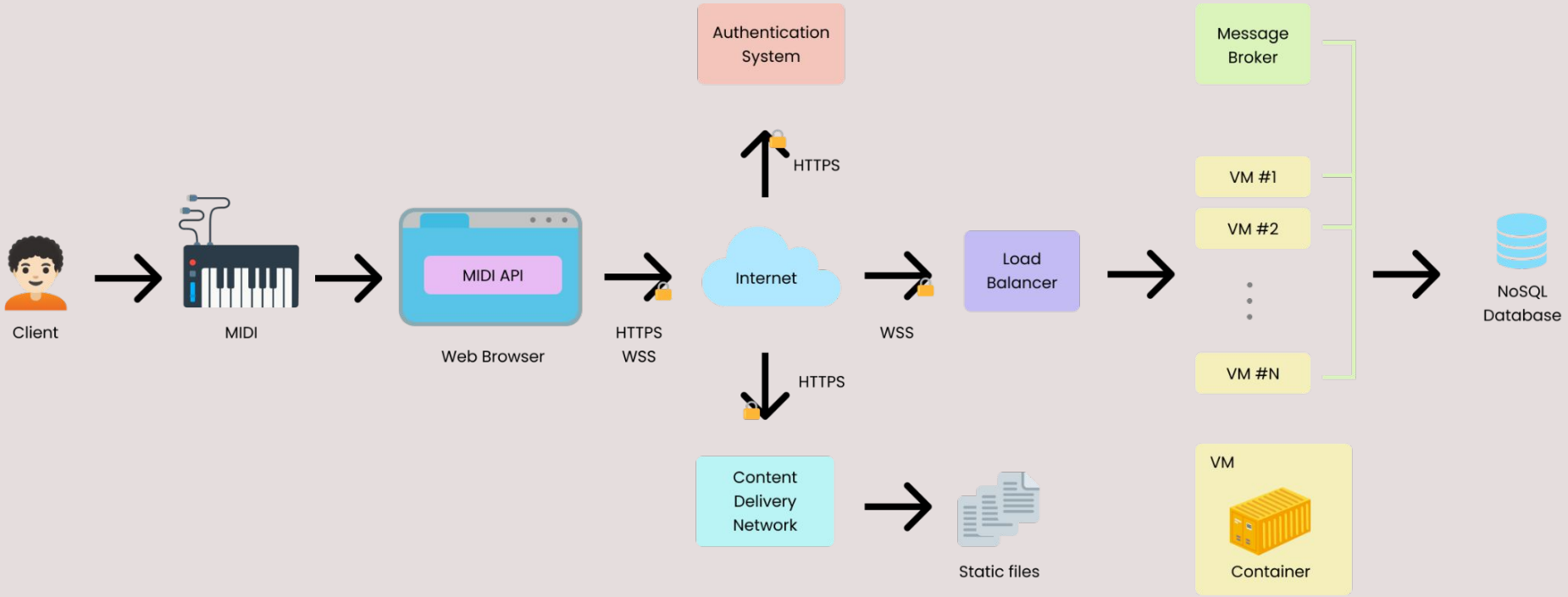
- **Web Application** based on cloud services
- Users can **play piano together**
  - In real time, using Web Sockets
- Virtual sessions called “**rooms**”
  - Notes played by one user will only be heard by users in the same room

User Interface

- Pressed **keys visualization**
- **Note visualization**
  - Scrolling rectangles
- Performance **recording**
- **Playback** and **looping**



Design - Architecture



Design - Server logic

- **Receive events** from Web Socket
- **Broadcast them** to users in the **same room**
  - **Problem**: possible only if users connected to same instance
  - **Solution**: message broker to share events
- Collect **latency** measurements
- Handle **recordings** operations with DB

Implementation - Front end

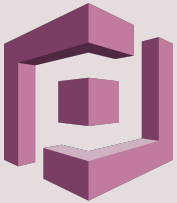
- **Javascript** for front end and back end
- **React** framework
- MIDI API (Chrome, Edge, Opera)
  - **Tone.js** and @tonejs/Piano
- **aws-amplify** for Amazon Cognito

Implementation - Back end

- **Node.js** runtime
- **Docker** engine
  - Dockerfile (e.g. DynamoDB table, port)
- Web Sockets (full-duplex, low latency)
  - **Socket.IO** (rooms)
- **Redis** for message exchanges
  - In-memory key-value storage
- Latency measurements (ping-pong)
- **aws-sdk** for DynamoDB

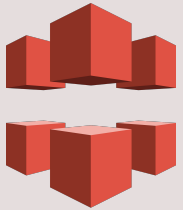
Deployment

- **Amazon Web Services (AWS)** with Educate account
- **Amazon Cognito** for authentication
  - User pool, client app ID
- **DynamoDB**
  - Recording table, on-demand capacity



Deployment - Front end

- Planned CI/CD with **CodePipeline** and **CodeBuild** (and S3)
  - Not available with Educate
- Front end needs to be served in **HTTPS** for MIDI
- **CloudFront** CDN
  - Generate certificate from ACM
  - Again both not available 😢
- Fallback: self hosted with self signed certificate

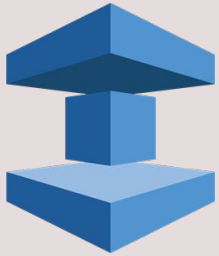


Deployment - Back end

- CI/CD with **CodePipeline**
- **ElasticBeanstalk**
  - Deploy web applications also using Docker engine
  - 1-3 EC2 t2.micro instances
  - Automatic scaling policy
    - Add instance if *Avg CPUUsage* > 66% for one minute
    - Remove instance if *Avg CPUUsage* < 20% for one minute
  - Load balancer
    - HTTPS: Custom signed certificate uploaded via IAM

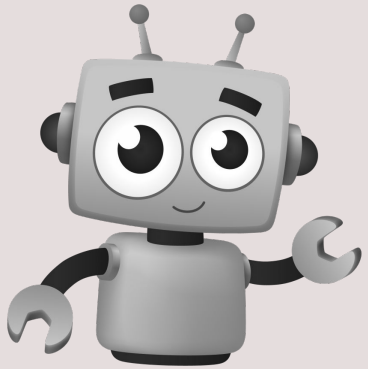
Deployment - Back end cont.

- Message broker
  - **AWS ElastiCache** - fully managed Redis
  - Listed but not available
    - Confirmed by AWS support
  - Self hosted for tests



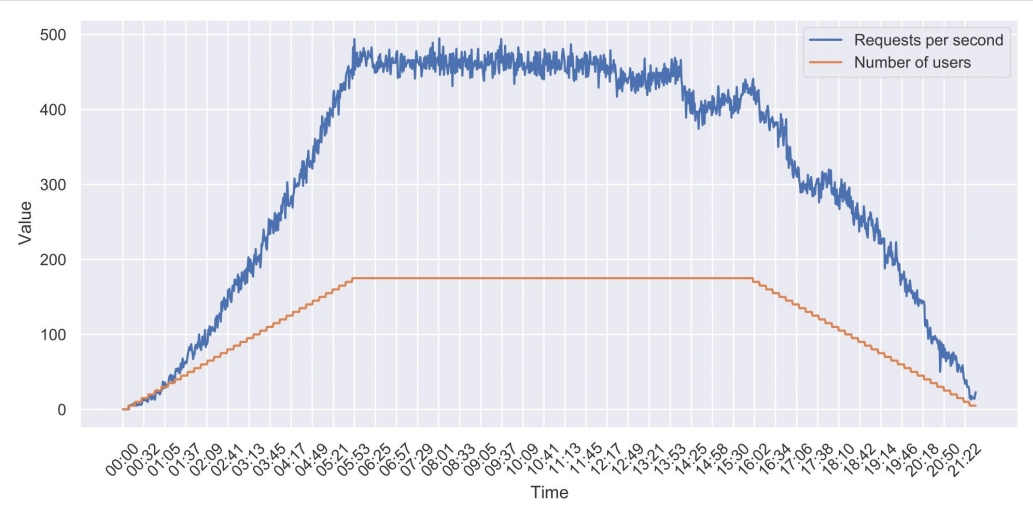
Validation

- **Custom Node.js script**
- Emulates **user behavior**
- Each user sends **random events** every **500ms**:
  - 90% Note events
  - 10% Database requests



Validation cont.

- **Ramp up**: 5 users every 10s
  - Max 175 users
- **Maintain load** for 10 mins
- **Ramp down**



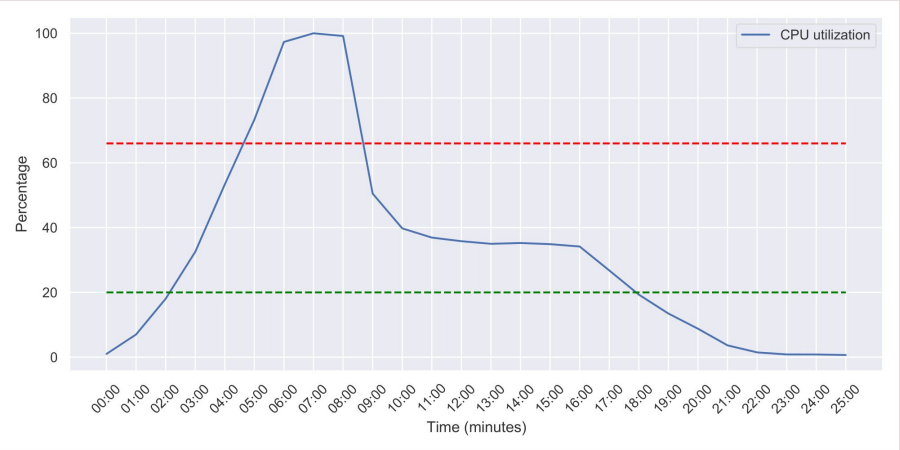
Validation cont.

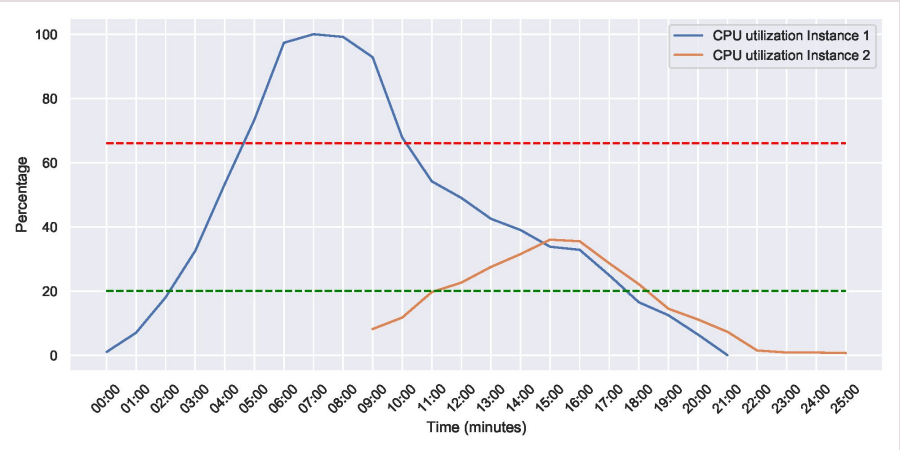
- **More complex** since we used Web Sockets
- **Problem:** when second instance starts, load not spread
  - First instance clog up due to **warm-up** and web socket **stickiness** (only first connection distributed)
- **Solution**: disconnect probability
  - Every second 1% chance to disconnect and reconnect
  - New connection handled by load balancer

- **CloudWatch** to collect metrics (.csv format)
- Enable EC2 **detailed monitoring** for more precision (5m ⇒ 1m)

Results

- 6:00: 100% CPU usage
  - **New instance** launched
- 9:00: New instance **ready**
  - Start **load balancing** (disconnect probability)
- 16:00: **Ramp down**
- 19:00: **Scale down**
- 21:00: Instance **shutdown**

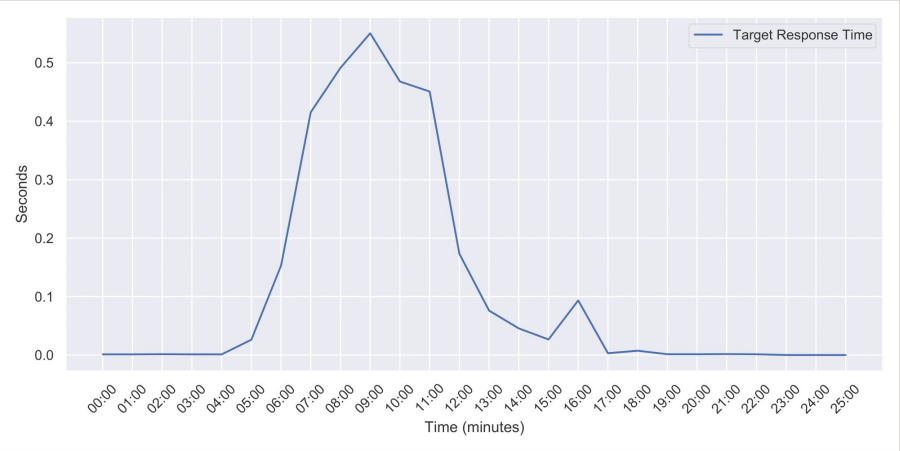


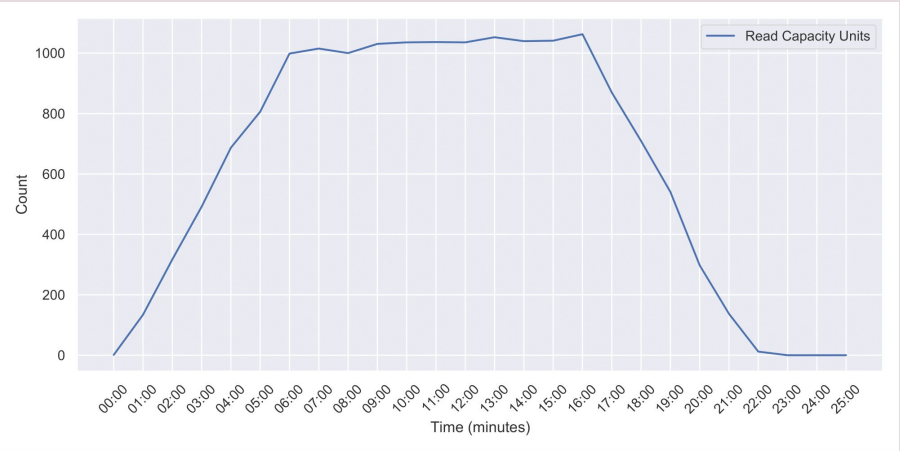


Results cont.

- **Response time** = seconds between request to LB and response from VM
- **Good** response time even near 100% CPU usage
- Always **< 600ms**

- DynamoDB **automatic scaling** with load in order to save money





Thanks for the attention!