Vrije Universiteit Amsterdam

MSc Thesis

Analysis of Failures in Massive-Scale Multiplayer Online Games

Author: Jakob Matthijs Kyselica (2593776)

1st supervisor:	Alexandru Iosup
daily supervisor:	Jesse Donkervliet, Sacheendra Talluri
2nd reader:	Animesh Trivedi

December 4, 2024

Contents

List of Figures

List of Tables

Abstract

Massive multiplayer online games (MMOGs) have become a big industry. As these games grow, so does the complexity of the underlying systems. These systems, distributed in nature, have the difficult task of supporting hundreds of thousands of concurrent players, all interacting with the virtual world. This complexity creates potential for failure. Issues such as degraded performance, or periods of complete unavailability can have a large impact on player satisfaction and therefore revenue. While the dependability of Cloud services is well-researched, studies in the area of MMOG dependability are rare, and the data sets required for such research are difficult to obtain. This study aims to alleviate these issues by presenting a system that allows for the easy gathering and analysis of relevant data sets. Furthermore, we present availability analysis of MMOG ecosystems ran by five different entities. This analysis shows that different MMOG ecosystems have similar failure characteristics, with failures generally occurring more than once a week, and on average lasting under half an hour, with most failures affecting only part of the active players. A large contributor to these outages was identified to be planned maintenance. This is an area where significant improvements to MMOG availability could be made.

1 Introduction

Massive multiplayer online games (MMOGs) have evolved into a large industry. The games industry is estimated to have had a total market value of $152bln in 2019 [24]. For reference, the global box office was worth $42bln in 2019. As of 2021, the top 5 MMOGs are estimated to have over 10 million active players [1].

MMOGs are a genre of online computer game where large numbers (often thousands) of players participate in and interact with the same virtual world. These worlds are often persistent, meaning that actions players take have a lasting effect on their in-game character and the virtual environment. Furthermore, MMOGs often contain complex virtual economies, and a strong social aspect, with players forming alliances to compete and cooperate in the game.

Today's large scale MMOGs, having their roots in 1980's adventure games and mainframe systems, began gaining traction in the 1990's, when early internet-based MMOGs started appearing. As these early MMOGs supporting handfuls of concurrent players have evolved into games serving hundreds of thousands of users, the complexity of the underlying systems has grown significantly. Today's MMOGs consist of distributed ecosystems that have the task of providing all players with a consistent high-quality experience, as MMOGs have been shown to be a QoS-sensitive service [5]. This is challenging, given the sheer complexity of the different components of the ecosystems and their interplay. Furthermore, there are high demands for consistency. An inconsistent game state can have a severe negative impact on players, for instance causing the loss of in-game assets. From performance degradation causing players to quit and move to other games, to complete outages of the service, modern MMOGs suffer from a plethora of potential failures.

Currently, game failures and their effects are not well understood, while, given how much revenue is generated with MMOGs, such failures can become costly. If a game does not deliver good availability and QoS, players will stop playing and move to a game with a better experience. It is therefore important to try to understand these MMOG failures.

The groundwork for online game-related research was laid in the Game Trace Archive [9], by designing a unified Game Trace Format, and providing a set of game traces collected by the authors. Such traces have been used in studies like [17], which performs analysis using player counts for the MMOG World of Warcraft, and [4], which does so for a range of games. These studies, however, do not focus on dependability. While the failures and dependability of cloud services is a well-researched area [25] [8] [14], studies relating these concepts to the field of MMOGs are scarce.

In this work we present the design of a game failure analysis data pipeline to facilitate the collection MMOG failure data, a method for the automatic detection of failures from player counts, and failure analysis using longitudinal data from two highly popular games.

1.1 Context

Modern MMOGs and the game services comprising them are faced with the task of supporting hundreds of thousands of concurrent players while offering good availability and QoS. To handle this, the total load on the system generated by the players needs to be distributed over multiple resources. We define *resources* as separate computing units such as physical or virtual machines, where communication between such resources is non-trivial

and must be explicitly addressed. To distribute the load, MMOGs use three parallelization techniques: zoning, instancing, and replication. Zoning splits the virtual game world into a fixed number of distinct areas, each handled by different resources. Instancing creates multiple distinct copies of (parts of) the game world, each handling a subset of the active players. Lastly, replication maintains a synchronized copy of the game world among the different resources, with each resource responsible for the processing of a subset of the players. These techniques allow game services to support multiples of the number of players that a single resource could support. Section 2.1 describes these techniques in more detail.

Data regarding the availability of game services is difficult to find. Large commercial game providers do not publish detailed availability information that could be used to perform in-depth failure analysis. Therefore, to perform such analysis of commercial MMOG ecosystems, data must be extracted from limited, publicly available sources.

The number of concurrent players active in an MMOG, a statistic commonly found in public sources, can serve as a metric that indicates issues with the availability of the service. Intuitively, if a game that usually contains thousands of active users suddenly has none, we can infer that some type of failure has taken place. However, given that modern MMOGs are heavily parallelized, failures occurring within these ecosystems most often do not affect all players at once. Therefore, to infer failures from concurrent player counts, more sophisticated methods that take into account changes over time have to be used. This can be achieved by periodically sampling the number of online players in a game service, and constructing a *time series*, allowing for the analysis of variations in the player count, taking into account the time domain.

1.2 Problem Statement

MMOGs are subject to frequent failures, which can have a large impact. Game companies stand to lose players, and therefore profits, while players can be subjected to a degraded experience, potentially lose valuable in-game assets, and decide to simply quit the game.

In a recent example, the space-themed MMORPG *EVE online* broke the world record for most concurrent participants in a multiplayer video game PvP battle, with over 6000 players engaging in the virtual war [10]. However, the ecosystem behind the game was not able to cope with the unexpectedly high load generated by the thousands of players all interacting in the same area, causing many players to lose their in-game assets, alongside the general unavailability of the service [7].

Another recent example of a large-scale MMOG failure can be found with Roblox, a popular MMOG based on user-created content. The game experienced a three-day long outage resulting in a complete loss of availability for all players [19]. The company describes the failure as the result of a combination of factors, such as heavy load and bugs in back-end service communications [3]. Minecraft Realms, the official online service for Minecraft, also recently experienced a large-scale outage [18] leaving players, who pay a monthly fee for access, unable to connect to the service for over twelve hours.

Although MMOG ecosystems fail regularly, there is a lack of scientific research exploring these failures. Such research has value for both academia and industry. For instance, research on MMOG architectures could benefit from analysis of failures prevalent in current MMOGs, to tackle these issues directly in new architecture designs. Industry can use

failure analysis to assess the failure behavior of their system in relation to other systems, and to aid in the prevention of failures.

1.3 Research Questions

This thesis aims to broaden the understanding of failures in online gaming ecosystems by analyzing longitudinal data from two highly popular games: *Runescape*, one of the largest free MMORPGs, and *Minecraft*, the best selling video game of all time.

To this end, we formulate the following research questions:

- **RQ1: How to collect availability data from game services?**
  Obtaining the data required to perform failure analysis of MMOGs is challenging. Game companies have little incentive to share their private data, and publicly available data sets are scarce. High quality data sets capturing data over long continuous periods of time are a fundamental requirement for research furthering the understanding of failures in real-world MMOGs. It is therefore beneficial to have a system that aids in the creation, analysis, and publishing of such data sets from limited sources, in a reliable, usable, and cost-effective way.

- **RQ2: How to detect game failures using publicly available data?**
  To obtain useful failure data from simple data sets containing the number of active players over time, failure events have to be marked. In large data sets, performing this classification manually is not feasible. Therefore, to enable failure analysis of longitudinal data, it is important to have a system that automatically identifies and marks failures in an active player count time series.

- **RQ3: What are the characteristics and statistical properties of online game service failures?**
  In an industry as big as the MMOG industry, failures can have large consequences. To understand these impactful failures, knowledge of their characteristics and statistical properties is valuable. This requires the construction of relevant and informative metrics, and thorough analysis and visualization of the available data.

1.4 Contributions & Approach

To answer these questions, this study presents the following contributions:

1. A design for a game failure analysis data pipeline.

   First, we present the design and system architecture for a data pipeline specialized in the collection, storage, and analysis of online game failure data. To validate the concept, we build a functional prototype containing the main elements of the design.

2. A method for the extraction of game failures from publicly available data.

   We present a method that allows for the automatic extraction of failure data from large data sets containing online player counts over time. We evaluate the performance of the algorithm using manually tagged sections of data.

3. Analysis and characterization of online game service failures over extended periods of time, including a comparison between different games.

   We analyze failure characteristics such as interarrival time, duration, and the number of players affected. Furthermore, we explore the consequences of failures, and the impact of planned maintenance on availability. Lastly, we compare the findings for the different analyzed games to reveal potential commonalities across games.

1.5 Outline

This thesis is structured as follows: Section 2 provides background information on the workings of MMOGs, general concepts regarding the dependability of such systems, and related work. Section 3 describes the design of a data pipeline focusing on the collection and analysis of game failure data, answering **RQ1**. To address **RQ2**, Section 4 presents a method for the extraction of failure data from publicly available online player counts. Section 5 answers **RQ3**, by presenting the analysis and results of failure data collected from two popular games. Section 6 discusses threats to the validity of this work, and finally, Section 7 presents the conclusion.

2 Background

The following section provides background information on the relevant principles and models behind MMOGs, and general concepts regarding failures in distributed systems.

2.1 Model of MMOGs

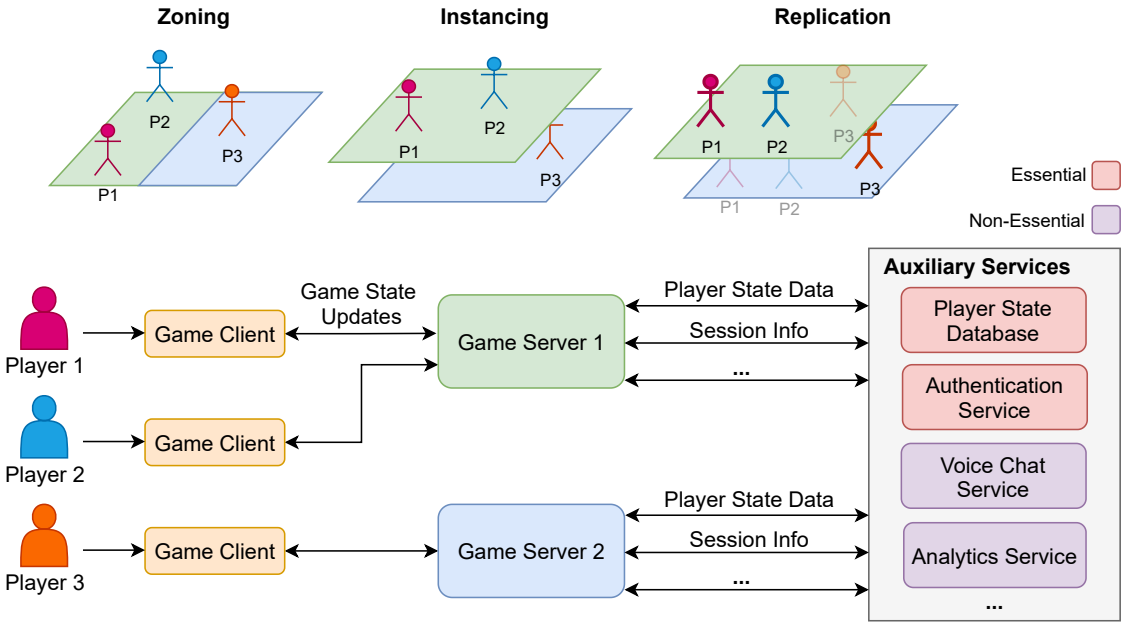

Figure 1: *Model of a generic MMOG architecture.*

MMOGs are most often based around a large *open world*. This means that players are free to explore and interact with the virtual world while choosing their own objectives, and are not guided through the game in a linear way. This makes the actions of individual players difficult to anticipate, and complex situations can emerge even from relatively simple game mechanics. The game world often represents large geographical areas with differing types of thematically distinct environments. Players have different reasons to be located in different parts of the virtual world, also referred to as the *map*. This means that players are not evenly spread along the entire map, with some areas attracting larger numbers of player than others.

Figure 1 shows a model of a modern MMOG architecture. The majority of current MMOGs operate on client-server architectures. Players use the client software to connect to a remote game server. This server sends messages back and forth, informing the client of the state of the game, and modifying this state according to input from the client.

One of the primary performance bottlenecks in situations with large numbers of players is synchronization. The state of each players has to be broadcast to every other player to maintain a consistent view of the game world. This causes a rapid growth in the traffic between the server and the clients, when more players are added. The extent to which the game world is synchronized between players is, however, flexible. For instance, a game may choose to not synchronize certain resources (e.g. mineable ores) in the game world,

causing each player to have their own 'view' of certain aspects of the world. This concept is occasionally used to the extreme, where a player gets a completely private 'copy' of a portion of the game world for certain passages of the game, without any synchronization with other players. Reducing synchronization means that some state changes caused by one player do not have to be broadcast to other players, reducing network traffic. However, it does potentially hurt the illusion of a consistent world which all players share.

In reality, reducing the level of synchronization to an acceptable level will not suffice for a single game server to be able to handle the amount of players required for MMOGs. The current practice is to solve this problem by distributing the load over multiple resources, using parallelisation techniques. The three main methods used for this are *zoning*, *instancing*, and *replication*, as depicted in the top row of Figure 1.

Zoning is a common parallelisation technique where the game map is divided into separate, non-overlapping areas called *zones*. Each zone is assigned to a separate resource (e.g. computation node), and does not interact with the other zones. This means that players that are situated in one zone can interact with each other, but cannot interact with players in other zones. Near the center of the zone, this is not a major limitation, as players largely only interact with their direct environment, which will be in the same zone. However, around the outer edges of the zone, this poses some limitations. Firstly, players looking over the edge of the zone will not be able to see the real state of that area, nor be able to interact with it. This is often solved by specifically shaping the game environment to exhibit natural boundaries (e.g. mountains or walls) where zones end, or by simply disallowing players to move over the zone boundaries. Secondly, players wishing to move from one zone to another will have to be transferred from one resource to another, causing an interruption of gameplay. These factors put a limit on the minimum size of the zones, and thus the number of zones the map can be split into while maintaining an acceptable gameplay experience. Furthermore, strictly using zoning raises issues of load balancing, where some parts of the game world are naturally busier than others, potentially causing undesirable load distribution over the available resources. This is why zoning is often combined with the next parallelisation technique; *instancing*.

Instancing entails the creation of multiple separate copies of the game world, each hosting a subset of the players and running on separate resources. The individual *instances* maintain their own game state, and do not communicate with one another, so players in one instance cannot interact with players in other instances. This makes supporting larger numbers of players straightforward, simply by adding more instances. Load balancing also becomes easier with instancing compared to zoning, as, from the perspective of the player, there is little difference in being in one instance over another. Instancing is often combined with zoning, to further increase the potential number of players. In this case, each zone has multiple separate instances. This increases player mobility between instances; the interruption inherent to inter-zone travel can be utilized to silently transfer players to different instances, according to the needs of the game operator. However, this introduces problems when players form groups, and expect to encounter the same players when moving to another zone. Each player in the group may be sent to a different instance of the zone they were travelling to, potentially causing frustration when the players can no longer find each other. This reveals the main downside to instancing; it limits the extent to which the game is truly 'massively multiplayer'. As each instance only holds a subset of the total number of players, each player can only interact with a fraction of the

total players at once, limiting gameplay possibilities, and harming the illusion of a single common world in which all players reside.

The last of our three main parallelisation methods is *replication*. Similarly to instancing, multiple separate resources maintain copies of the same part of the game world. However, whereas instancing treats these copies as autonomous and does not allow interaction between them, replication works around this problem by synchronizing the game state among the *replicas*. Each replica serves a subset of the players, and processes only the actions of the players connected to it. The results of the players' actions are subsequently broadcast to all other replicas. This means that each replica holds a copy of the full game state, while only processing part of it. The entities that a replica is responsible for are called *active entities*, while the entities whose state is received from the other replicas are called *shadow entities*. This method of parallelisation addresses the main issue with instancing, where players across different instances were not able to interact. Replication allows for player across all replicas to interact, while still allowing the game to scale up the number of concurrent players. However, it does introduce some issues regarding the synchronization of game states between players across different replicas. In some situations, such as when two players from different replicas attempt to pick up the same item, the two replicas have to communicate to decide which player actually receives the item. Without this, the item could end up duplicated or both players could end up without it. This inter-replica communication could introduce additional latency, negatively impacting the gameplay experience.

Regardless of parallelisation, some aspects of the game need to remain consistent across all resources. For example, the player's character data needs to be carried over from instance to instance, so items collected in one instance will still be in the player's possession once the player is moved to another instance. This means that all the parallelised game servers still need to communicate with a single system; the player database. Many such auxiliary services are used in today's gaming ecosystems. Some of these are of a non-essential nature, such as a voice chat service or a back-end player analytics service. While failure in some of these services may cause a QoS degradation, the game itself will remain functional. Other auxiliary services are essential to the game's operation. The authentication service, for instance, validates a player's credentials when attempting to connect to the game. If this service fails, players will be unable to connect to the game, regardless of the status of the actual game servers.

2.2 Concepts regarding dependability of distributed systems

Given how modern online gaming ecosystems are essentially large-scale distributed systems, they suffer from the same issues that affect other types of distributed systems. Throughout this work we use the basic concepts and definitions regarding system dependability presented in [2]. Distributed systems are susceptible to a range of possible problems, including hardware failures, configuration errors, traffic overloads, and natural disasters. These problems may or may not result in system **failures**; *events where the system ceases to operate according to its specification.* For instance, the failure of a single hard drive within a data center may not produce any effects that are visible from the outside of the system. However, a natural disaster will most likely cause a service failure. Parts of the system state that may lead to failures are called **errors**, and the root cause

of an error is a **fault**. For the purposes of this thesis, we focus on failures that render the system unavailable to users. We call these periods of unavailability **service outages**.

Given the nature of distributed systems, service outages are not binary; users in one region of the world may not notice any problem with the system availability, while users in another geographical region may have completely lost access to the system. This means that distributed system outages may vary in severity, with most outages not affecting all users at once.

2.3 Related Work

Research in the field of availability of MMOGs is rare. However, work exploring concepts closely related to this study is available. The Game Trace Archive [9] presents the design of a unified game trace format, with similar design requirements and considerations as the system presented in Section 3. Similarly, the Failure Trace Archive [14] introduces the design for a unified failure trace format for the archival of failure traces from various distributed systems. The design in Section 3 uses concepts found in both, creating a system specifically suited to the collection and storage of MMOG failure data.

In the field of MMOG research, [17] explores characteristics of active player count variations over time for the game World of Warcraft, similar to how failures are inferred in this thesis. [4] performs a similar characterization of the number of active players over time, for multiple games, including weekly patterns and long-term trends. However, these studies do not use this data to deduce failures.

While failure analysis of MMOGs is novel, failure analysis can be found in research on other large-scale distributed systems, such as the cloud. The aforementioned Failure Trace Archive presents failure analysis of various distributed systems investigating the duration of availability and unavailability intervals. Similarly, [25] analyzes failures found in high-performance computing systems by exploring failure rates, time between failures, repair times and root causes. [15] also uses real-world data to investigate the availability of enterprise desktop grids, exploring characteristics such as availability and unavailability intervals, and failure rates.

Lastly, [26] collects user reports on cloud failures and characterizes them empirically. The study uses publicly available data on failures to allow for analysis of real world (commercial) systems which do not divulge in-depth data regarding their operation and failures. This is similar to this thesis, where the lack of failure data on real world MMOGs is mitigated through the use of publicly available active player counts.

3 Designing a Game Failure Analysis Data Pipeline

To work with and to analyze failures in online multiplayer game systems, one needs access to data containing such information. Currently, such data is difficult to find as game companies have little incentive to publish detailed information on their own service failures. The data that *is* publicly available is stored in many different formats, complicating its usage in research. This section answers **RQ1** by presenting both a system architecture that manages the collection, storage, and analysis of online game availability data, and a unified storage format for said data.

3.1 Design Requirements

The system presented in this section consists of three broad functional requirements.

FR 1: Data Collection. As publicly available data sets on the availability of online game systems are rare, the system must be able to gather this data. This can happen in two ways: the system can scrape raw data containing availability information from public sources such as web pages, and the system can accept data being 'pushed' into it by third parties, such as a game company donating its data for research purposes.

FR 2: Data Storage. The data coming into the system must be stored in a unified way. This facilitates the usage of the data contained within it, as it allows for a single set of analysis tools to work across all data sets available in the system. The unified storage format for online game availability data used in the system is described in Section 3.3.

FR 3: Data Processing and Analysis. The data stored up to this point is likely in a raw, unprocessed state. To be able to analyze this data, the system must be able to process it. For instance, scraped web pages must be parsed to extract the desired data. This processed data can then either be directly taken from the system for manual analysis, or used in automatic analysis tools provided by the system.

Alongside the functional requirements discussed above, we have set up several non-functional requirements for the system.

NFR 1: Elasticity. The system has to be elastic. This means that it must be designed in a way that allows the system to scale up or down its utilized resources according to demand. This is helpful in cloud environments, where such behavior can both decrease operational cost by scaling down in periods of low demand, and allow for the possibility of scaling up to quickly meet increased demand when necessary. For instance, if a cooperating party is offering real time metrics, the system might need to be able to accept very large amounts of push requests with valuable data while minimizing the impact on the data provider. The system must be able to provide a quick response to the data provider, with further processing delegated to a set of workers that can grow the processing capacity with minimal bounds. To prevent stale data, the scrapers in the system must acquire their data as close to the designated time point as possible, and the system must be able to support a growing number of scraping jobs in a quasi-linear way, by growing the number of workers.

NFR 2: Extensibility. The system has to be easily extendable. As some public data source may present data in a way that is not yet supported in the system, it has to be simple to add new data gathering and processing units, without interfering with other parts of the system. To further aid extensibility, the implementation of the worker units should not be constrained by a single programming language, and workers written in different programming languages should be able to communicate over the same channels.

NFR 3: Data Archival. The data used in this system is of a fleeting nature. For example, the active player count of an online game at a certain moment in time may only ever be available at that specific moment. If one were to discard this raw data after having parsed the desired values, data could be irreversibly lost. For instance, the structure of a web page may change during the scraping process. The configured parsers could then fail to extract the desired data, resulting in the loss of the original raw data as well. For this reason, the raw incoming data should always be stored as-is, allowing processing and analysis to be applied multiple times, and to historical data. The data should be stored in a way that allows for partitioning, and distributing the data over multiple physical resources.

3.2 System Overview and Architecture

Figure 2 shows an overview of the data flow of the system. Data flows through the system from top to bottom, and the user interacts with the final stage of the data flow; the service section. Firstly, data is gathered from different sources, such as scraped web pages or incoming pushed data **(1)**. After validation, the data is stored in the *data lake* **(2)**. To ensure no data is lost and to facilitate data repair, the data lake has two separate sections; *good data*, and *bad data*. The validation step decides in which section the incoming data is stored. Raw data is taken from the data lake and processed by the processing workers, such as parsers **(3)**. The processed data generated by the processing workers is stored in the *data warehouse* **(4)**. Lastly, the service section **(5)** is responsible for the communication between the user and the system. It allows users to manually control the data processing section, request aggregated data reports, and configure data sources. The following subsections discuss the architecture in more detail.

3.2.1 Communication and Messaging

To address **NFR 1** and **NFR 2**, the system is based around *message queues*. Firstly, message queues allow for easy inter-process communication. This means that the components of the architecture can be split into separate processes, which can run on separate resources. Furthermore, the usage of a message queue for the communication between different system components means that these components do not have to be written in the same programming language. Any program using the chosen message queue protocol can plug into the system architecture.

The message queue-based communication is primarily used to orchestrate the workers performing jobs in the system. Figure 3 shows how these jobs flow through the different parts of the system. There are two job queues; the *ingest job queue* **(2)**, and the *processing job queue* **(3)**. Jobs are pushed into these queues through two possible means. They can

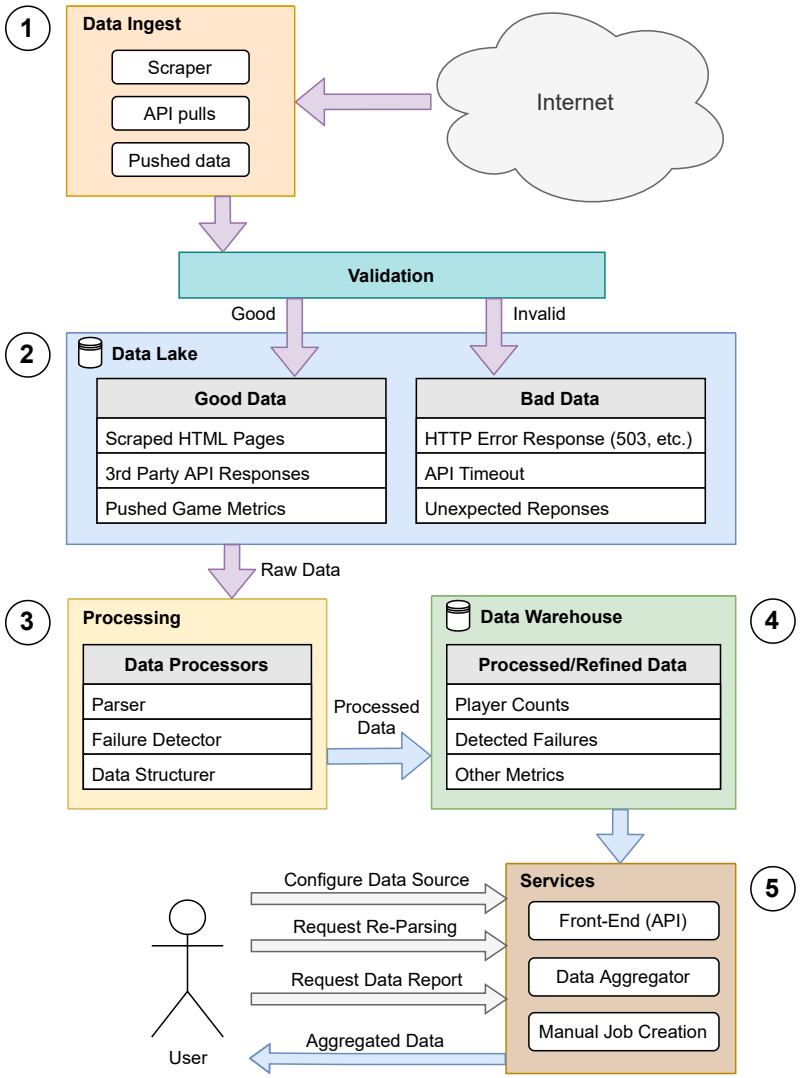

Figure 2: *Overview of the data flow of the system.*

be automatically generated by a scheduler, such as an ingest scheduler **(1)** that creates a scraping job for a particular web page once every two minutes, or they can be manually created by a user. For instance, a user could manually create a parsing job that will parse a specific date range of scraped web pages **(4)**. Once a job enters its corresponding job queue, an available worker from the worker pool **(5)** can take it from the queue and perform the job. Each type of worker is capable of performing a specific job, and only listens for those types of jobs in the job queue (e.g parser workers **(W4)** will only pick parsing jobs from the job queue). There can be an arbitrary number of worker types, and each worker type can have an arbitrary number of running instances. This allows for the number of workers to be dynamically scaled, addressing **NFR 1**. For example, if it is observed that the majority of the load on the system comes from parsing jobs, it is simple to run more parser workers to cope with the higher load. Likewise, less stressed

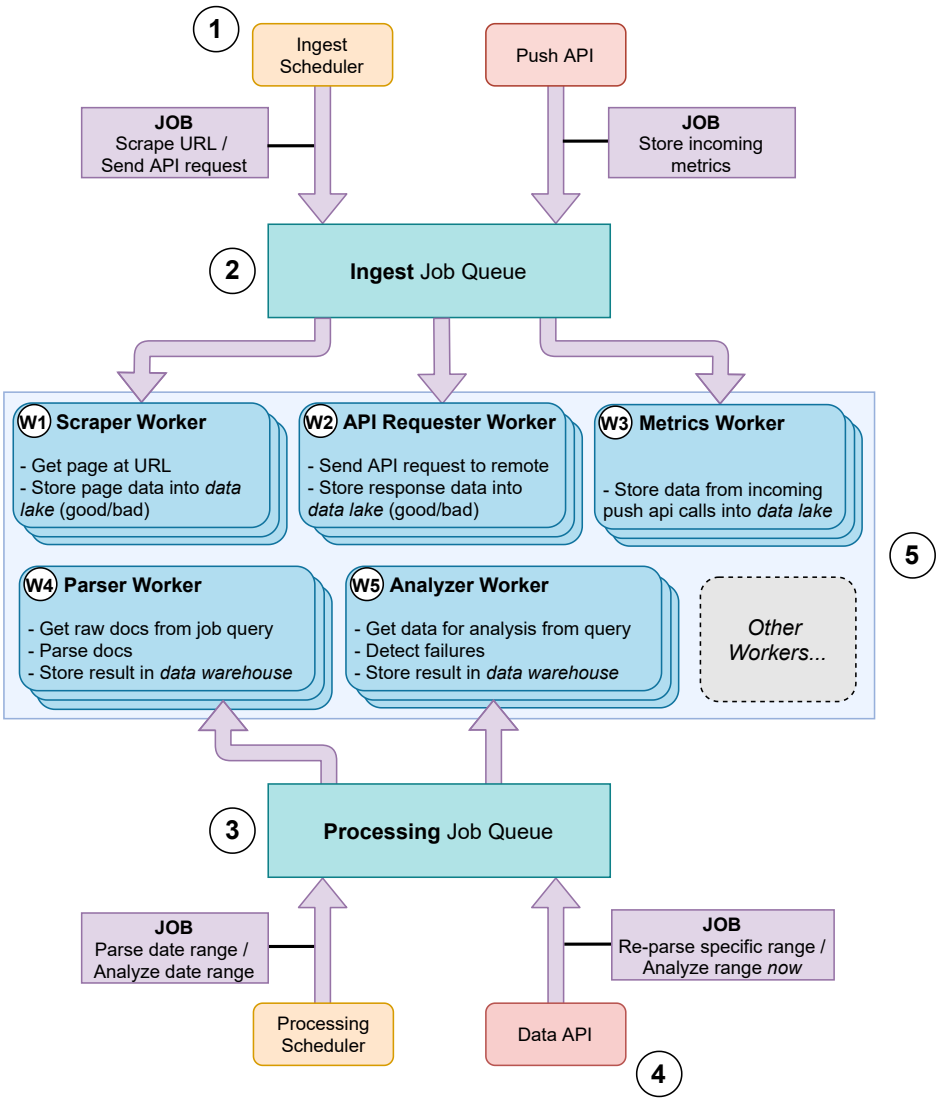

Figure 3: *Overview of the message queue-based architecture.*

components of the system can be scaled down to operate with fewer workers. This makes the system suitable for cloud environments, where dynamic resource scaling can increase cost-effectiveness.

While communication within the system is managed through message queues, communication with external actors is handled through HTTP-based APIs. The system contains two APIs, the *admin API* and the *push API*. These APIs are separated and capable of running on different resources, as the typical workload and traffic signature of both APIs differs. The admin API is responsible for the retrieval of data from the data warehouse requested by the user. Furthermore, it allows the user to manually create jobs within the system. For example, to re-parse a specific time range of scraped web pages. Configuration tasks, such as the management and addition of new data sources, are also handled through the admin API. The push API allows the system to ingest data that is proactively

pushed into the system by some actor, for instance, a game operator wishing to keep track of some statistic. This data is received, validated, and stored in the data lake.

3.2.2 Data Collection

Data can enter the system in various ways. Aside from the aforementioned push API, two additional sources are considered in this design. The first, web page scraping, allows for the periodic scraping of web pages containing relevant data. For instance, the home page of an online game may contain the current number of active players. In the system configuration, this source can be added by defining the URL to be scraped, and the schedule on which to perform the action. Once the source is configured, the *scheduler* component will start creating scraping jobs using the pre-defined schedule. These jobs will be picked up by the scraper workers, which will perform the actual fetching of the web page, and will store the resulting data in the data lake.

The second additional data source is pulling data from external APIs. Some services may publish relevant information, such as player counts, not through web pages but through some form of API. For instance, Minecraft servers can be polled for their current online player count. To enable the system to ingest data of this type, API pull workers work alongside the scraper workers. Their jobs are, again, created by the *scheduler* component, and defined by the user in the system configuration.

3.2.3 Data Processing

The data processing section of the system consists of a set of data processing workers, which take either from the data lake, or the data warehouse, and store their resulting processed data in the data warehouse. The parser worker is an example of such a worker. This worker takes data that was scraped by a scraper worker and stored in the data lake, and parses the desired data out from the stored web page. This is done according to the data source configuration created by the user, which also houses the scraping configuration. Here, the user specifies the desired HTML element containing the data. The contents of this element can then be further filtered using a regular expression, to extract only the relevant data. This processing is performed on a schedule, which can be separate from the scraping schedule, as it may be desirable to perform the processing by batches (e.g. process all files of the previous day, once a day). The extracted data resulting from this process is stored in the data warehouse using the unified format defined in Subsection 3.3.

Another type of processing worker is the failure extraction worker. This worker takes data containing player counts from the data warehouse, and extracts failures from it. This process is described in detail in Section 4. The process results in a list of extracted failures from the date range specified in the job, also using the unified storage format.

3.2.4 Data Storage

The system contains two data stores. The data lake, and the data warehouse. The data lake contains a large amount of raw data that does not have to be accessed frequently. It functions as a kind of archive, to ensure that no usable data is lost within the system. Conversely, the data warehouse contains processed, ready to use data that may be fre-

quently requested by users. Specifically, it contains clean time series containing player counts, and lists of failure events.

This split was chosen to address **NFR 3**. As noted, it is important to retain all gathered data, because it is of a fleeting nature. However, keeping all of this raw historical data in the same place as the processed, operational data would introduce performance and cost issues. We therefore choose to keep these two data stores separate.

3.3 Game Failure Storage Format

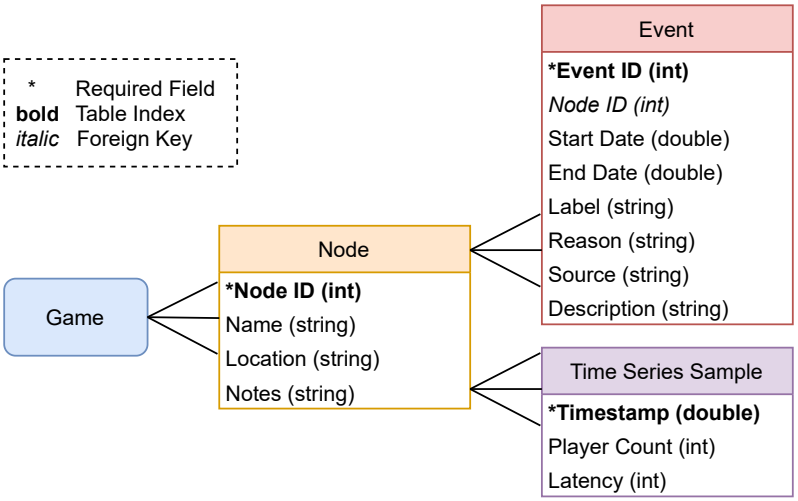

Figure 4: *Structure and individual fields of the game failure storage format.*

The game failure storage format presented in this subsection aims to facilitate the storage, exchange, and analysis of online game failure data through the use of a unified format. It is designed to support many different types of online games, and is designed around the most common publicly found availability metrics: online player count and latency. The design draws from related trace storage formats found in the *Failure Trace Archive (FTA)* [14] and the *Game Trace Archive (GTA)* [9], combining concepts found in both systems into a single format that is uniquely suited to the storage of game failure data gathered from public sources.

The format focuses on availability data in the form of online player counts. This allows it to have a simple and strict structure, by not needing to account for varying types of data. This strict structure allows analysis tools to assume the presence of certain fields, and to work across all data sets stored in this format. Despite this, the format can support the storage of data from a large range of games, as the number of online players is a general statistic that is relevant to most online games, and in the case where game failures are obtained through means other than player count analysis, this data can still be adequately stored (i.e. online player count time series are not required to be able to store data on game failures).

FTA, while supporting the storage of failure events similar to the ones found in this format, is not designed to enable the storage of the time series, which are at the core of the failure detection used in this thesis. *GTA* does support the storage of custom data

formats (including time series), however, there is no pre-defined structure for the types of data we are storing, leaving the necessity for a common format.

Figure 4 shows the structure of the storage format. Each field is marked with the expected data type. There are four types of entities: games, events, nodes, and time series samples. Each game is comprised of a set of nodes, which act as universal abstractions for gaming traces. These nodes could, for example, represent instances or zones in an MMOG. This level of abstraction was chosen for its simplicity, while retaining the level of detail typically used in failure analysis. For instance, if one only has access to the global player count over all instances of an MMOG, the format can accommodate this data simply by using a single node. If more fine-grained data such as per-instance player counts are available however, one can store this data under multiple nodes. The only required field for nodes is the **Node ID**, used for linking events to nodes. This was chosen to support scenarios where minimal information regarding the structure of the analyzed system is known.

Each node has a series of samples which record the online player count and latency of the game for that particular node. In essence, each node has its own time series containing availability measurements. Time series are a simple way of storing periodic measurements, which are at the core of this format. Storing these measurements line by line with a timestamp also allows for missing data and potentially varying sampling frequencies. As recorded data may only contain one of the two types of measurements, neither the player count nor latency field is required. Additionally, this allows for samples with no measurement values, in case a fixed time difference between each sample is desired. Each time series associated with a node is stored in a separate table/file, as opposed to storing the time series samples for every node in a single file, distinguished by a **Node ID** field. This allows for analysis of the measurements on a per-node basis, without needing to filter the data (i.e. the files contain one measurement per row, and can be directly imported into data analysis tools as a single time series). Furthermore, the time series files are simplified by not needing a **Node ID** field.

Events represent known failure events of the investigated system. These can be automatically generated from, for instance, the player count time series, or can be manually created. The only required field for events is the **Event ID**. The other fields are not required, as partial knowledge of events can still be useful to store. For instance, one may observe a failure that occurred at a known time, with a known cause, but no clearly defined end time. This is potentially useful data that would not be storable in a format that, for instance, required both a start and end date. In addition to the start and end date, events can contain a label signifying the type of failure, the reason for the failure, the source of the information, and a description. Each event can be linked to a node using its **Node ID**. An empty node field in an event can signify a global event, or other types of events where the node is not relevant.

The described structure is independent of storage method. The entities could simply be stored as *CSV* files in folders, or the entities could be saved as tables in a relational database. It is up to the specific implementation to decide which method is most suitable in that scenario.

3.4 System Implementation

As a proof of concept, we have implemented a working prototype of the design described in the previous subsections. It is written in Node JS, with RabbitMQ as the message broker. The main components of the system are the scheduler with its main configuration, the workers, and the two APIs. All of these components are separate applications that can run independently of one another. This addresses **NFR 2**. When a new type of worker is required, it is simple to add a new worker application without needing changes to other parts of the system. Furthermore, it is simple to run more instances of a worker to increase load capacity, as the message broker will handle the division of tasks between the active workers.

The configuration file contains the definitions of the data sources used. Each source has its own list of associated jobs. For instance, a source could be 'Runescape', with two associated jobs: scraping the Runescape homepage, and parsing the scraped data. Each job has its own schedule and job parameters. For example, the scraping job is scheduled to scrape once every two minutes, and the job parameter is the URL of the page to be scraped. The parsing job would have its own schedule, and its parameters would include the specific HTML element to extract the desired number from. The scheduler uses this configuration file to create new jobs, according to the specifications.

All components use a common *storage model*, which means that the way in which files are stored can be changed easily. Currently, data is being stored in the file system, but a switch to database oriented data storage would only require a change in the storage model.

The failure detection worker is written in Python, due to Python's excellent data science tools. This also showcases the flexibility of the architecture, where the only constraint for a worker is the ability to communicate through the message queue.

In addition to the components mentioned in the design, the prototype also includes a simple graphical user interface to facilitate communication between the user and the system, shown in Figure 5. This graphical front-end, written using React, uses the admin API to control the system and request data. It allows the user to select a date range and be shown a graphical view of the time series and, optionally, the previously detected failures within this time range.

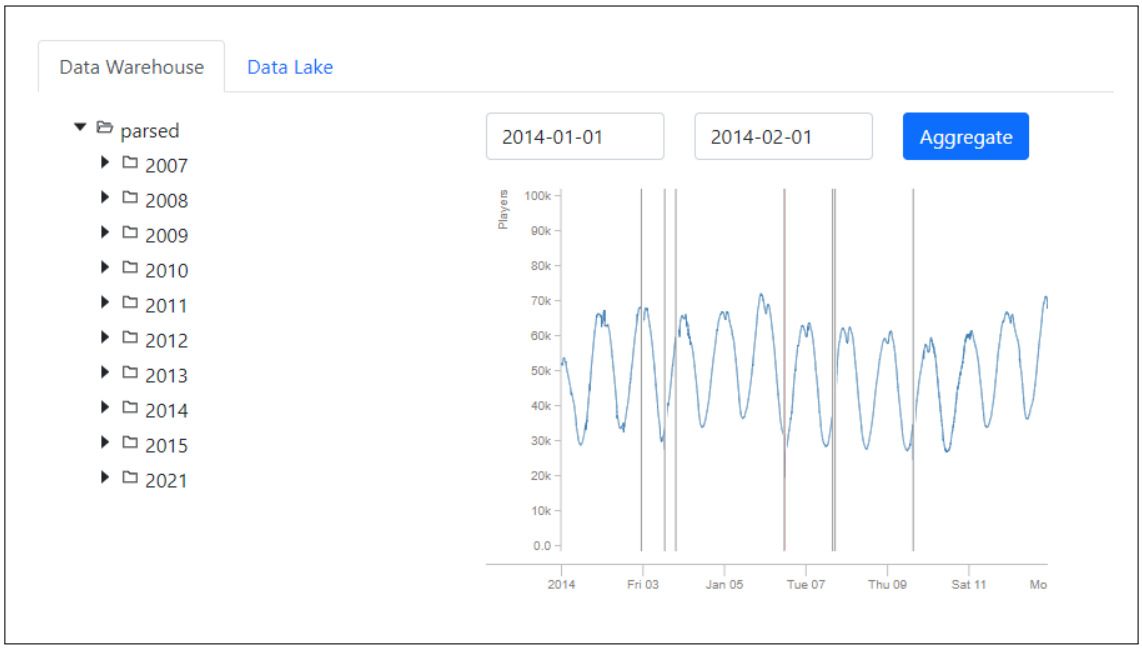

Figure 5: *Screenshot of the prototype GUI.*

4 Method for the Detection of Failures from Player Counts

To identify and analyze failures in online gaming ecosystems, we look at their online player counts and how these numbers evolve over time. Anomalies in this data, in particular where the player count suddenly drops sharply, can give insight into how these game services fail.

As these failures can occur suddenly and last mere minutes, it is important to use player count data with a sufficient sampling frequency in the order of one sample every few minutes. However, given that failures usually happen at frequencies in the order of days, it is also important to have data spanning a sufficiently long period of time (at least several months). This presents the problem that it may not be feasible to manually go over all the data and pick out the anomalous parts. Therefore, this section describes a method for the automatic detection of failures from high-frequency player count time series, answering **RQ2**.

4.1 Description of MMOG Datasets

The datasets used in the following sections all consist of a time series containing the number of connected players at each time point. This data is sourced from two different MMOGs, and is summarized in Table 1.

Dataset	**Source Game**	**Date Range**
ds-runescape	Runescape	August 2007 - March 2015
ds-hypixel	Java Minecraft - *Hypixel*	August 2020 - May 2021
ds-minehut	Java Minecraft - *Minehut*	August 2020 - May 2021
ds-cubecraft	Bedrock Minecraft - *Cubecraft*	August 2020 - May 2021
ds-thehive	Bedrock Minecraft - *The Hive*	August 2020 - May 2021

Table 1: MMOG datasets summarized.

4.1.1 Runescape

Runescape is one of the largest free-to-play MMORPGs, with an estimated total player count of 2.6 million in 2015 (the last year captured in the dataset). It was originally released in 2001, and uses *instancing* to support large numbers of concurrent players. The dataset containing the total online player count for Runescape, **ds-runescape**, spans from August 2007 to March 2015. It was collected by scraping the Runescape homepage [12], which features a self-reported total online player count, using a sample period of 2 minutes. Figure 6 shows an excerpt from the raw dataset. To obtain a usable time series from this data, we parsed each HTML page to extract the active player count. This number is highlighted in bold in Figure 6.

4.1.2 Minecraft

Minecraft, the best-selling video game of all time, has spawned a large ecosystem of independent third-party service providers allowing players to partake in multiplayer activities.

```
<div class="header-top">
    <ul class="header-top__right">
    <li class="header-top__right-option">
        <strong id="playerCount">66,797</strong> Online
    </li>
    <li class="header-top__right-option">
        <a class="header-top__right-link" href="...">
            Old School
        </a>
    </li>
```

Figure 6: *Excerpt of raw data from the Runescape data set.*

The larger of these multiplayer service providers, referred to by the community as 'servers', are run by companies with their own business models, separate from the game's developer Mojang. These 'servers', often offering their own 'minigames' and services built on top of Minecraft, operate on principles similar to MMOGs such as Runescape, where the multiplayer element is handled by the game operators themselves. Parallelization is achieved through instancing and zoning.

We use 4 different datasets spanning the same period of August 2020 to May 2021. Each dataset represents the online player count for a specific Minecraft service. We selected the 2 most popular services for both the PC-based Java edition, and the cross-platform Bedrock edition of the game. In order, these are: Hypixel, Minehut, Cubecraft, and The Hive, resulting in datasets **ds-hypixel**, **ds-minehut**, **ds-cubecraft**, and **ds-thehive**, respectively. The original dataset sourced from MineTrack [16] was collected through periodic requests of the self-reported player count, and has a sample period of 3-5 seconds. However, we resample this data for a final sample period of 2 minutes, to aid processing times.

4.2 Data Preprocessing

As we established, the primary form of data used in this thesis are time series. The first plot in Figure 7 shows an example of how this raw time series data looks. Firstly, we check whether there is a seasonality, meaning a recurring pattern with a fixed period. If such a pattern exists in the data, it means that the fluctuations caused by that pattern are not anomalous. For instance, more people may play games in the evening than the morning, causing every evening to have a higher value than the following morning. To facilitate finding the points which fall outside of this regular pattern, it is useful to remove the seasonal component from the data. To this end, we first use an autocorrelation plot to find the seasonalities. This shows a very clear 24-hour seasonality, with a less prominent 7-day seasonality. Secondly, the data may contain a long-term trend, which the seasonality follows. Observing the raw player count time series shows the clear presence of a trend.

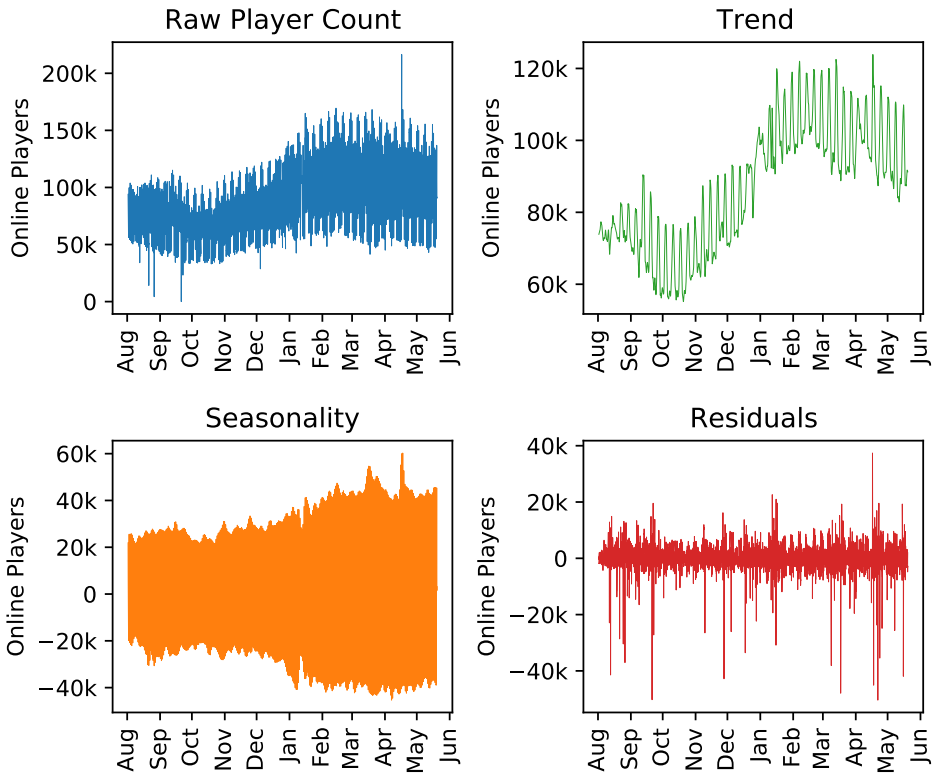

Figure 7: *Seasonal decomposition of the dataset from Java Minecraft service Hypixel (**ds-hypixel**).*

It is beneficial to remove this component from the data, so that all fluctuations revolve around the same point, and the absolute value of points can be compared across the entire time span.

To extract and remove these two components from the data, we use STL [21]. STL, standing for "Seasonal and Trend decomposition using Loess", is a time series decomposition method that consists of a sequence of applications of the loess smoother. It decomposes the time series into the trend component, the seasonal component, and the remainder. This remainder is useful for the identification of anomalies, as it contains the deviations from the regular patterns. STL was chosen because of its support for arbitrary sampling frequencies (unlike X-13ARIMA-SEATS which only supports monthly and quarterly data), and because its simplicity allows for fast computation; something which is rather important given the high-frequency data which, in the case of Runescape, has over 2 million data points.

The result of the decomposition using STL can be seen in Figure 7. The first chart shows the raw unprocessed player count data. The second chart shows the trend component of this data, which follows the general shape of the raw data. The third chart shows the seasonal component, which shows the daily pattern present in the data. The fourth and

final chart shows the residuals; what is left of the data after the trend and seasonal components have been removed. As can be seen, the fluctuations in the residuals revolve around the 0-point. Additionally, larger and clearer negative spikes can be seen in the residuals when compared to the raw data. These are of importance for the next subsection.

4.3 Detection Algorithm

The chosen algorithm for the classification of anomalies uses, at its core, two different methods which both find largely disjoint sets of anomalies. The first method calculates z-scores over a rolling window, operating on the deseasonalized dataset. When a data point is N standard deviations below the mean of the window, it is marked as an anomalous point.

The second method uses the first order difference of each data point in the raw dataset. If this first order difference for a data point is sufficiently large, it is marked as anomalous. In essence, this means that there was a sharp drop in player count within the span of one sample. This method is performed on the raw dataset, as the seasonal components do not interfere in the calculation of the first order difference. The points marked as anomalous by the two aforementioned processes are shown in Figure 8 as dark green dots.

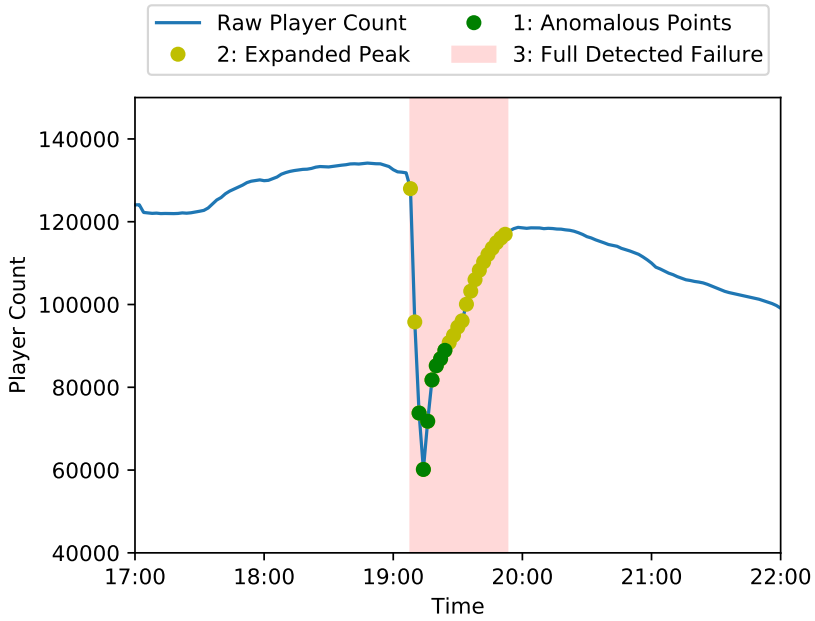

Figure 8: *Visualization of the failure detection procedure showing points marked as anomalous and the fully captured peak.*

Both of the aforementioned methods mark specific individual points as anomalous, but these points may not cover the full span of the failure. To ensure that the entire observable anomaly is captured, the anomalous points are expanded backwards and forwards in time. For each point marked as anomalous, its preceding points are evaluated one by one until a sharp drop in the first order difference on the raw value is observed. The threshold value

Param.	Name	Description
W	Window size	The number of samples in the window used to calculate z-scores.
N	Z-score threshold	The minimum z-score necessary to mark a point as anomalous.
D	First order difference threshold	The minimum difference between two neighbouring samples for the points to be marked as anomalous. This is expressed as a fraction of the mean value of the data set, as the magnitude of the first order difference is an absolute quantity, dependent on the data set mean.
S	Peak start difference threshold	The minimum decrease in first order difference required to mark a point as the start of a peak. This is expressed as a factor. For instance, an S of 4 would mark the start of the peak at the point where the first order difference becomes 4 times lower than the previously checked point.
R	Peak end recovery %	The required percentage of the pre-peak player count that needs to be regained for the peak to be marked as finished.

Table 2: Parameters of the algorithm used to detect failures from player counts.

of this difference is a parameter (S). This stopping point is defined as the start of the peak. Subsequently, a similar process is performed on the other side of each point. In this case, the stop condition is given by two possibilities. Either the player count has recovered to R% of the player count before the anomaly, or the player count starts to drop again before reaching R%. This is then defined as the end of the peak. All data points between the start and the end of the peak are then marked as anomalous. The points added by this step are marked in Figure 8 by the olive colored dots. The parameters of the described algorithm are shown in Table 2.

Figure 9 visualizes these parameters. The plot on the left shows the parameters relating to the marking of anomalous points (**W, N, D**). The window marked in green represents the window of the moving average, with its width represented by **W**. The dotted green line marked by **N** represents the z-score threshold. Points under this threshold are marked as anomalous. Lastly, the magenta line marked by **D** represents the first order difference threshold for marking points.

The plot on the right visualizes the parameters related to the expansion of the peak (**S, R**). Parameter **S**, shown by the blue arc, shows the peak start difference threshold; the minimum decrease in first order difference required to mark the start of the peak. Parameter **R**, shown by the brown arrow, represents the percentage of players that need to be regained in order to mark the end of the peak. The red arrow shows 100% of the original player count, for reference.

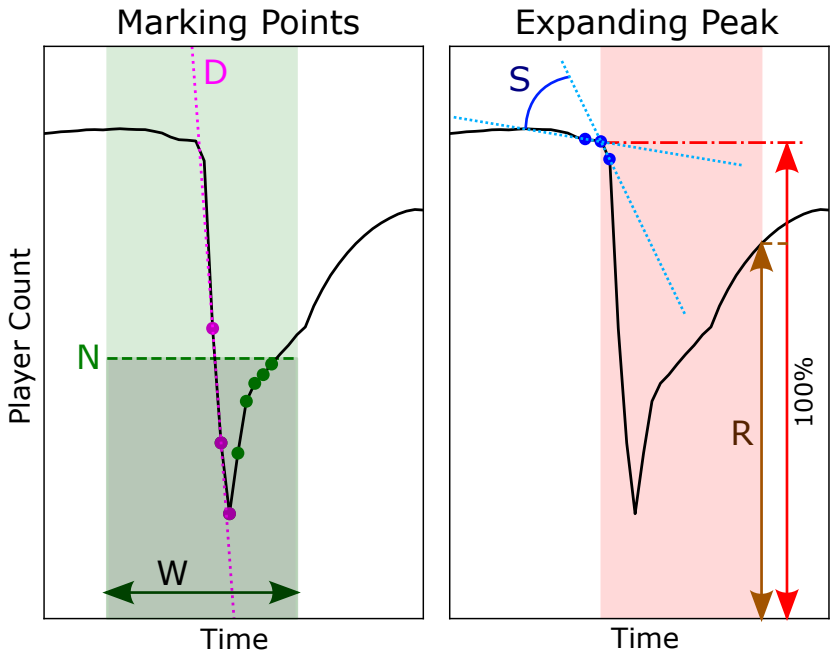

Figure 9: *Visualization of the parameters of the peak detection procedure. Displayed parameters are described in Table 2.*

4.4 Evaluation & Parameter Optimization

To evaluate the performance of this algorithm, we have manually tagged part of the data, allowing for various metrics to be compared between different algorithms and parameters. This not only allowed us to choose the best performing algorithm for this task, but also allows for parameter optimisation of this algorithm.

The parameters were optimised using a tree based regression model. The data exhibits a large class imbalance, meaning there are far more points which are *not* part of a peak than there are points which *are* part of a peak. This entails that metrics such as *accuracy* (fraction of correct classifications) do not properly portray the performance of a classifier, as simply classifying each point as *not part of a peak* would yield a very high accuracy. For this reason, the metric we used for scoring is Cohen's Kappa [6], which is robust against large class imbalances.

We manually tagged both **ds-hypixel** and **ds-minehut**. We performed k-fold cross validation on both data sets, with k=5. This resulted in an average kappa of 0.69 for **ds-hypixel**, and 0.62 for **ds-minehut**. The final parameters we chose based on this evaluation, and use in the remainder of this work, are as follows:
W = 4784, N = 4.86, D = 15.3, S = 2.06, R = 0.84.

4.5 Resulting Data

The aforementioned process results in a dataset with each sample labeled either as part of a peak or not. To extract the individual peaks and their characteristics, contiguous chunks of samples marked as part of a peak are grouped into individual peaks with their

characteristics. These are: duration, magnitude, and drop. The duration of the peak is calculated by subtracting the time of the first point in the peak from the time of the last point in the peak. The magnitude of the peak is given by the difference between the point with the highest value and the point with the lowest value. The 'drop' property is related to the magnitude; it is the magnitude expressed as a percentage of the highest value of the peak. This means that a drop of 100% signifies that the player count dropped to 0, and a drop of 50% implies that half of the online players were disconnected.

5 Results and Analysis

This section presents the results and failure analysis of the data for Runescape and four Minecraft services, answering **RQ3**. Our main findings are as follows:

O-1: Failures occur frequently, with the highest average failure interarrival time across all analyzed games being 5 days.

O-2: The duration of the failures tends to be short relative to the typical duration of a game session, with average failure durations under 30 minutes.

O-3: Failures do *not* tend to occur more often in busy periods.

O-4: Failures generally affect less than half of the active players, and complete failures (where more than 95% of active players are affected) are rare; in all cases, less than 18% of failures could be classified as complete.

O-5: The duration of a failure and the fraction of affected players show a positive correlation. Longer failures tend to affect a larger part of the active players, and conversely, shorter failures tend to affect fewer players.

O-6: On average, a drop in active players can be observed following a failure, even after the failure has been resolved.

O-7: Planned maintenance is a large contributor to service outages, with 34.8% of failures for Runescape being related to maintenance.

O-8: The analyzed games exhibit similar failure characteristics, with the strongest differences showing in the temporal patterns of the failures.

O-9: Updates in Runescape tend to occur when the player count is around the lowest for that day.

5.1 Overview of Analyzed Data

This subsection briefly discusses and visualizes some of the features of the the analyzed data, such as long-term trends. We validate the data by cross referencing visible events with reports from the game operators, finding significant concordance.

5.1.1 Runescape

To gain an overview of the available data for Runescape, we plot the instantaneous online player count over the entire period, starting from October 10th 2007 and ending with March 22nd 2015. Figure 10 shows this, with the vertical axis showing the player count, and the horizontal axis depicting time. The blue line shows an exponential moving average, representing the long-term trend in player count. Immediately, some large events are observable. Around the beginning of 2011 (marked by the first vertical dotted line), a sharp increase in player count can be seen. This coincides with the removal of free trade restrictions within the game's economy, causing many players to return to the game [22]. The drastic drop in player count in October of 2011 (marked by the second dotted line) also coincides with a significant event in the game; the so called "Bot Nuking Day" [23], where large amounts of automated players (commonly known as 'bots') were removed from the game at once.

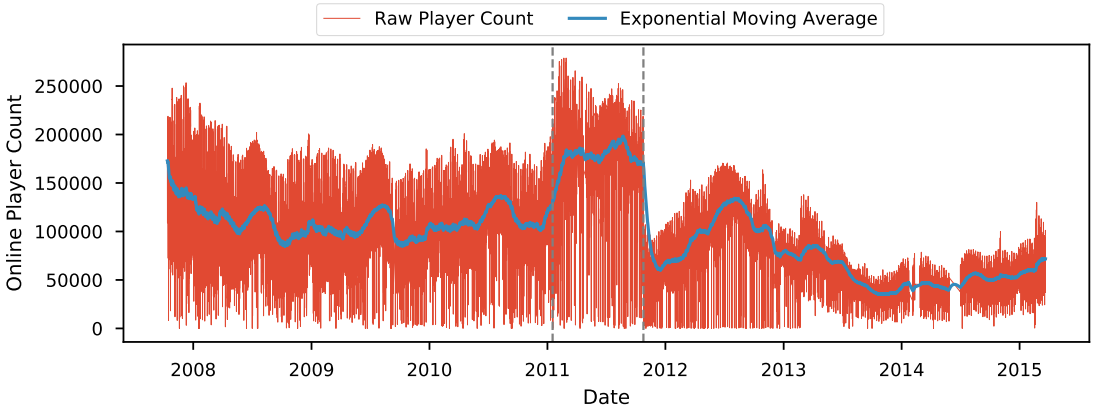

Figure 10: *Total online player count over the span of 7 years for Runescape.*

To further validate the data, we look at the month of July 2014, and cross-reference the observed player count dips with statements from the game developer and operator Jagex. This section of the player count plot is shown in Figure 11. For 6 out of the 10

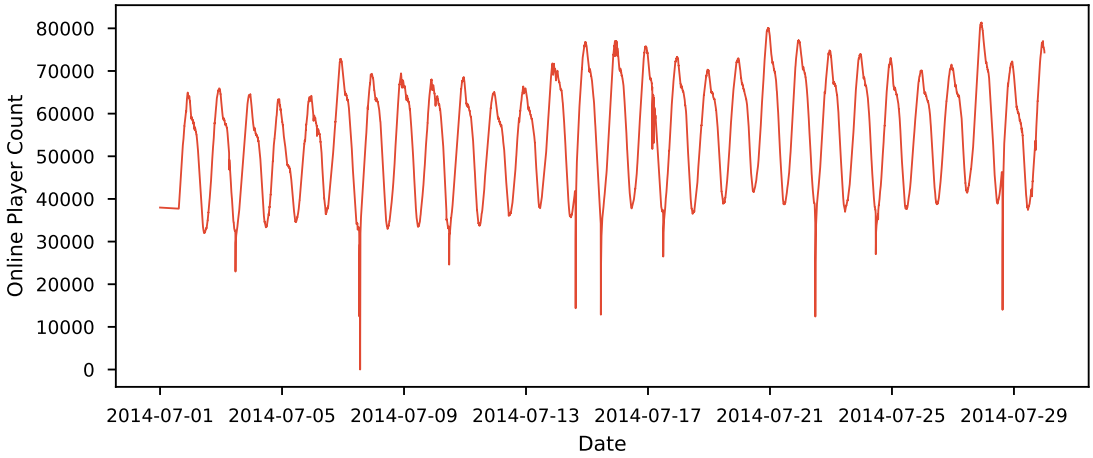

Figure 11: *Total online player count for Runescape during July 2014.*

visible dips, Jagex specifically mentions a game update as their cause. The remaining 4 player count drops had no comment. Interestingly, these updates always occurred when the player count was around its lowest for that day **(O-9)**.

5.1.2 Minecraft

Figure 12 shows the total online player count over time for the 4 selected Minecraft services: Hypixel, Minehut, Cubecraft, and the Hive. Hypixel shows a fairly steady trend in player count, with the average player count hovering between 65,000 and 100,000 and only a handful of clear outliers. Its player count makes it the most popular Minecraft service of the four, by a substantial margin. Minehut shows more variation over the time period, growing from an average of around 6,000 online players in October of 2020 (first gray

dotted line) to approximately 18,000 online players in February of 2021 (second gray dotted line). Furthermore, there is an apparent disruption in the trend around the beginning of December 2020, with a large increase followed by a sharp decrease in online players. Both of the Bedrock-based Minecraft services, Cubecraft and The Hive, show a similar trend. These disruptions are marked purple in the figure. Starting from August 2020, the two Bedrock-based services appear to have a slowly declining average player count, until the inflection point in the middle of November 2020 (marked with a lime green dot) is hit. For both services, this point marks a sharp increase in player count, after which it does not return to the values before the inflection point. Both services share a similar average player count.

To validate this data, we look at all self-reported outages from Hypixel from October 2020 to May 2021 [11]. 9 out of the 13 outages mentioned by the company are also clearly observable in the data. These are marked in Figure 12 with a green line.

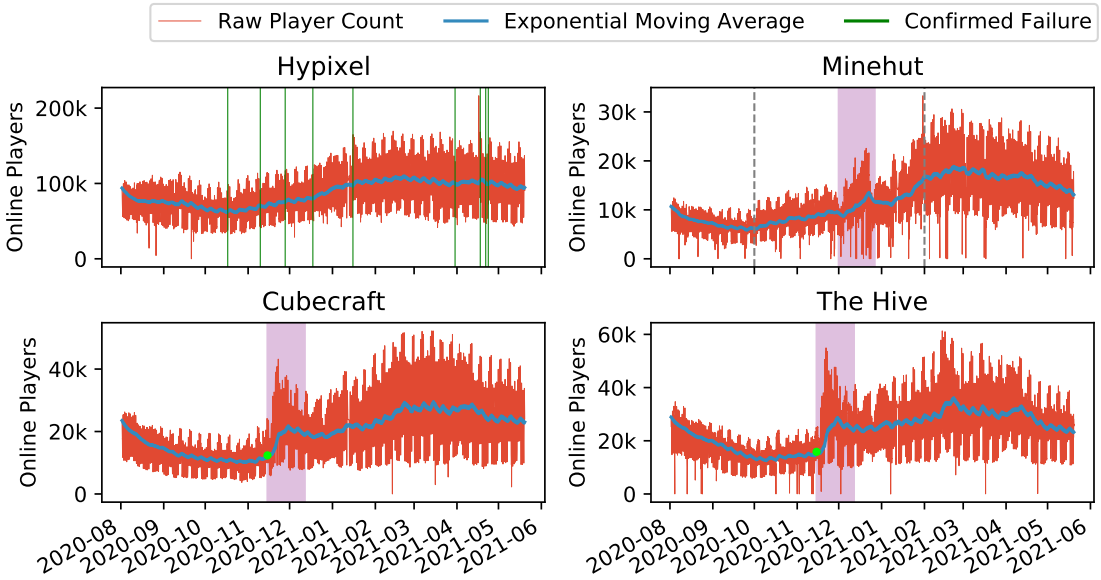

Figure 12: *Total online player count over the span of 9 months for 4 different Minecraft services.*

5.2 Characteristics of Online Game Failures

This subsection discusses the characteristics of the failures observed in Runescape and Minecraft. To gain insight into the failures occurring in these games, we explore their failure count, durations, interarrival times and scale. Furthermore, we investigate whether more failures generally occur during periods of higher traffic. We find that failures occur frequently, with none of the analyzed games achieving an average failure interarrival time of under 5 days (**O-1**). The failures tend to be short, with average failure durations under 30 minutes (**O-2**), and the average failure affecting less than half of the active players (**O-4**). Lastly, we see that failures do not seem to occur more often in busy periods (**O-3**).

Num. Failures	2341
Avg. Duration	21 minutes
Avg. Interarrival Time	1 day
Avg. % of players affected	41.7%

Table 3: Summary of failure statistics for Runescape.

5.2.1 Runescape

Table 3 shows a summary of the basic failure statistics for Runescape. A total of 2341 failures were detected, with an average interarrival time of 1 day. The maximum interarrival time was found to be 46 days. In terms of duration, we found an average failure duration of 21 minutes, with 41.7% of the players being affected on average. To get a more detailed look at the distribution of the durations and interarrival times, Figure 13 shows the empirical cumulative distribution function (ECDF) of these two metrics. We see that the interarrival times seem to follow an exponential distribution until the inflection point at around 7 days. This suggests a long tail, where 98% of interarrival times are shorter than 7 days, with only a few failures reaching up to 46 days. In terms of duration, we see that 95% of failures are shorter than 1.5 hours.

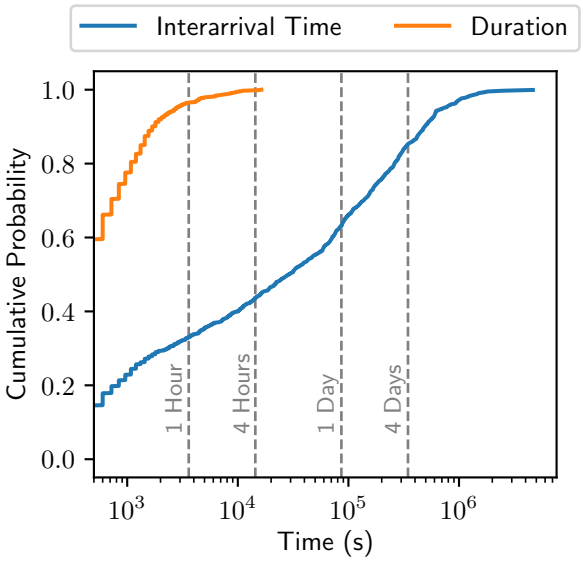

Figure 13: *ECDF of failure duration and interarrival time for Runescape.*

To determine whether more failures occur during periods of high traffic, we look at Runescape's daily traffic pattern and its failure pattern. Figure 14 shows these patterns side by side, with the upper heatmap showing the average player count for each hour of every day of the week, and the lower heatmap showing the total number of failures that happened in each of these time slots. Interestingly, the heatmap of failures does not seem to correspond to the heatmap of player counts (**O-3**). The most populated area (evenings

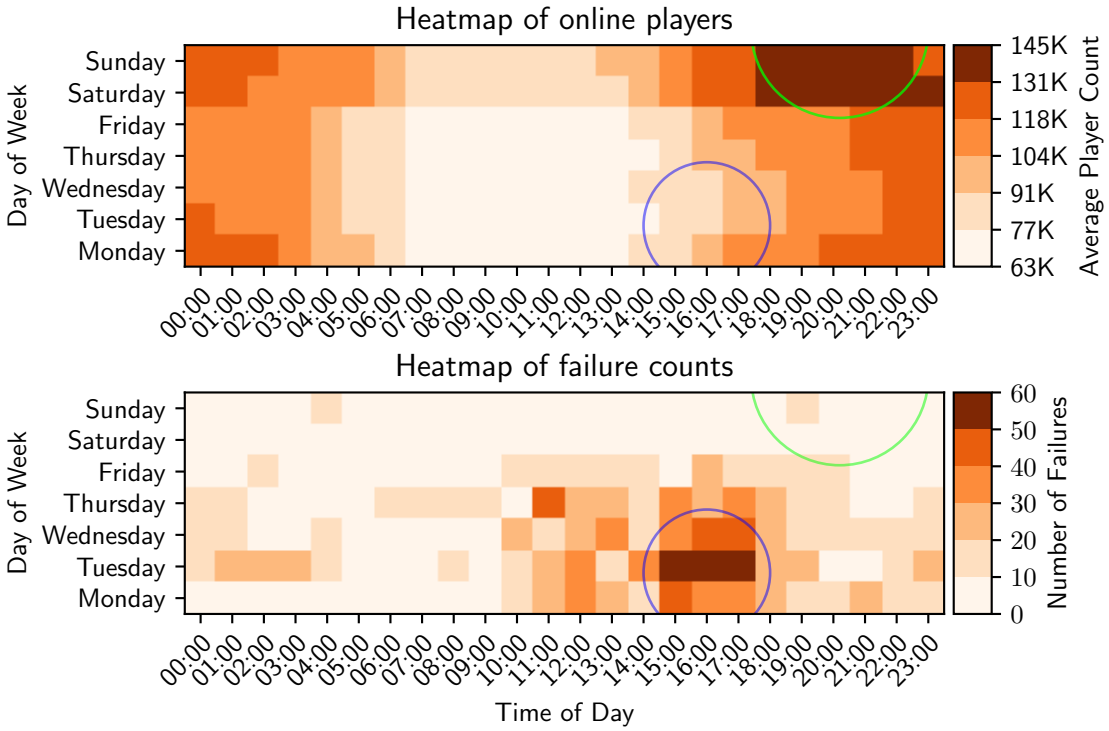

Figure 14: *Average number of online players & failure density heatmap for every day of the week for Runescape.*

of the weekend, shown by the green circle) has seen very few failures compared to the area around 16:00 on Tuesdays, with the surrounding timeslots also seeing large numbers of failures (indicated by the blue circle). This mismatch between failures and number of active players holds true when accounting for planned maintenance, which is further explored in Section 5.5.

5.2.2 Minecraft

Table 4 shows a summary of the failures of the four selected Minecraft services. Hypixel has experienced a total of 91 failures, whereas Minehut has experienced a total of 275 failures. The two Bedrock-based services have experienced fewer failures, with Cubecraft and The Hive experiencing 52 and 73 failures respectively. The average failure durations

	Hypixel	**Minehut**	**Cubecraft**	**The Hive**
Num. Failures	91	275	52	73
Avg. Duration	20 minutes	24 minutes	32 minutes	28 minutes
Avg. Interarrival Time	3 days	1 day	5 days	4 days
Avg. % players affected	26.3%	40.6%	27.3%	37.2%

Table 4: Summary of failure statistics for Minecraft services.

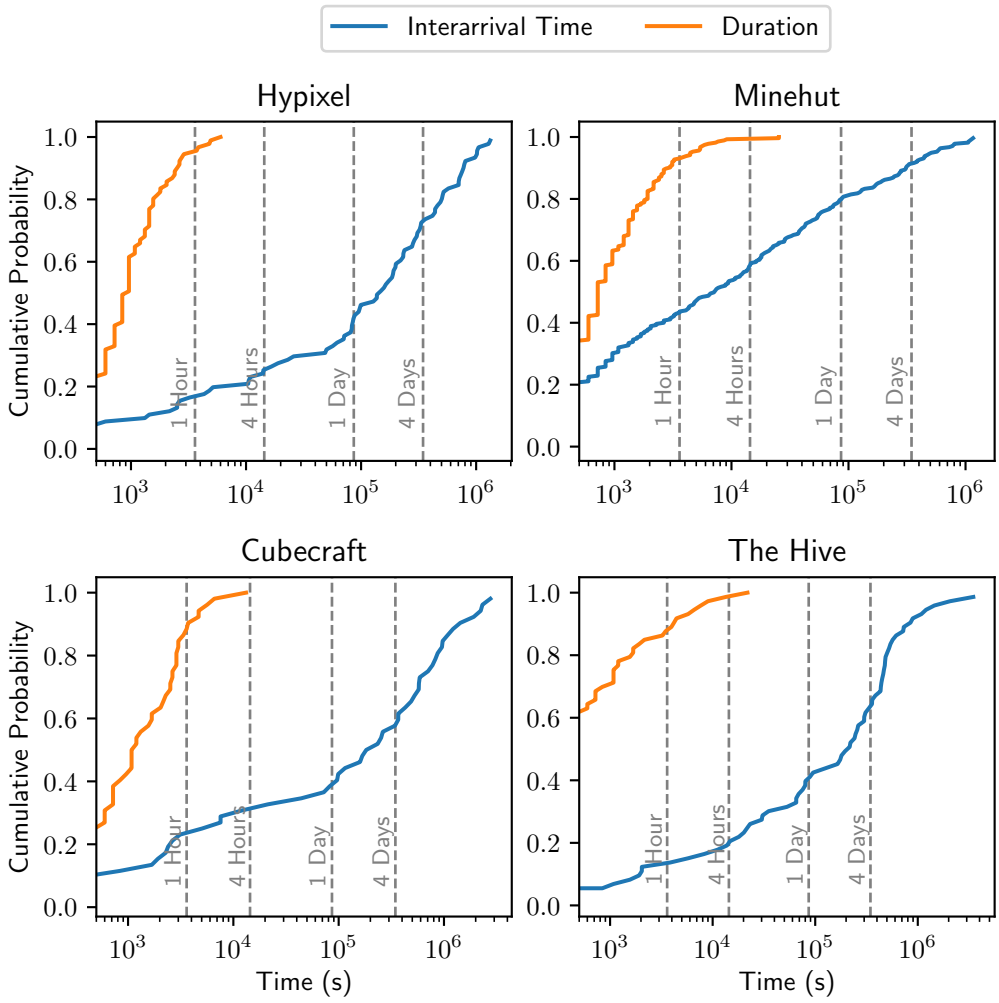

Figure 15: *ECDF of failure duration and interarrival time for 4 Minecraft services.*

are similar for all four services, with a mean failure duration of 26 minutes across the services. Figure 15 shows the ECDF plots of failure duration and interarrival time. The vertical axis represents the cumulative probability, and the horizontal axis represents the time in seconds. Firstly, we see that Hypixel, Cubecraft, and the Hive have a similar interarrival time distribution, while Minehut's differs significantly. This is confirmed by performing a Kolmogorov-Smirnov test between all pairs of services, yielding an average value (expressing the distance between the two distributions) of 0.15 between Hypixel, Cubecraft and the Hive, and an average value of 0.43 between Minehut and the other services. All services exhibit a maximum interarrival time of longer than 13 days, with the Hive and Cubecraft reaching maximum interarrival times of 40 days and 32 days respectively. This shows that while failures occur frequently, occasional long periods of correct operation do also occur. Regarding duration, we see that 95% of failures across all services are under 1 hour and 20 minutes, making the longest failures rare.

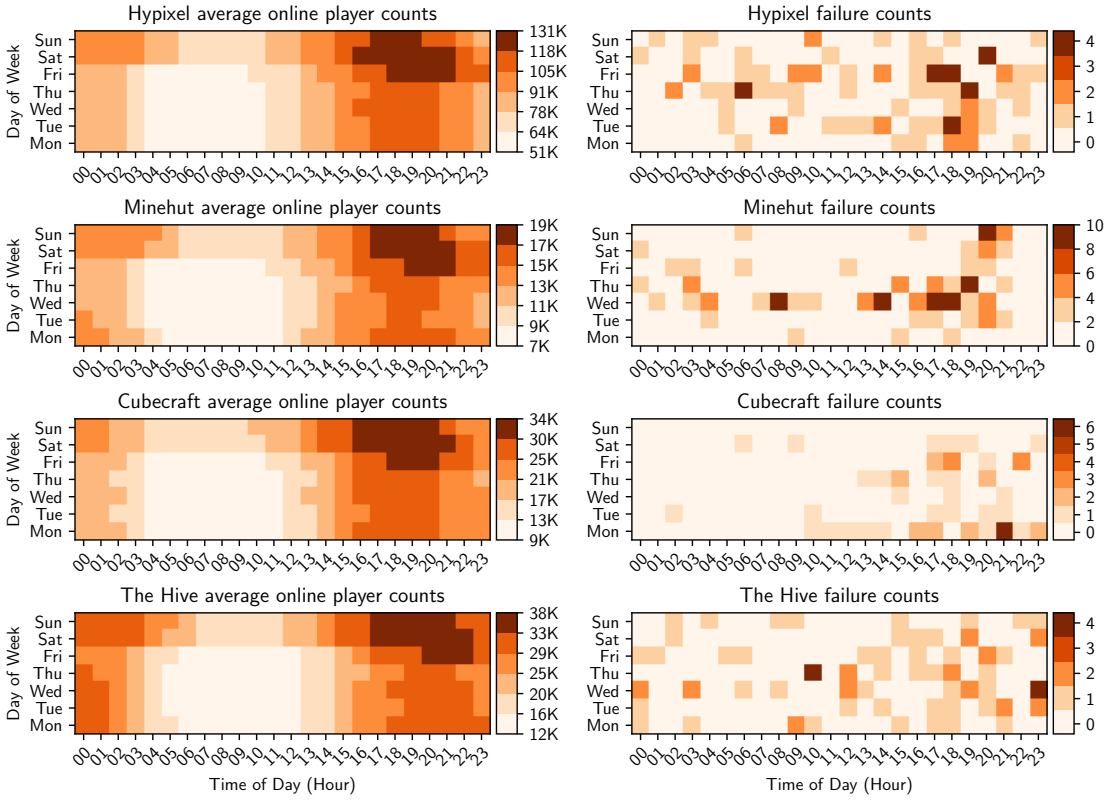

Figure 16: *Average number of online players & failure density heatmap for every day of the week for 4 Minecraft services.*

Figure 16 shows, on the left, the average number of online players at each hour for each day of the week. The right side of the figure shows the number of failures that occurred within each of these 'timeslots'. The time is referenced to UTC. Firstly, we see that all 4 services exhibit a very similar traffic pattern. The evenings are the most crowded, and the weekend shows an overall increase in players over the whole day. In terms of failures, it is interesting to note that there is only a partial overlap between highly populated moments and high numbers of failures. Evenings do appear to not only have the highest amount of online players, but also have the most failures. However, most failures do not occur on the weekend; the most crowded time. (**O-3**)

5.3 Number of Affected Players

The distributed nature of MMOG architectures results in many potential points of failure. For instance, any of the distributed resources could suddenly crash. The effect of this resource failure depends on the task this resource was responsible for. If, for example, the resource was responsible for a specific *zone*, this zone would become inaccessible, with players residing in that zone having to be transferred, resulting in a *total interruption of gameplay* [20]. However, the rest of the game would remain operational.
Some parts of the ecosystem are more critical, however. If a failure were to occur within

the authentication system, this could bar users from accessing any of the game servers, effectively rendering the entire game unavailable, regardless of the status of the game servers themselves. The following subsections explore these aspects in terms of the fraction of players that are affected by the failures. We find that complete failures, where nearly all players are affected, are rare. The Hive exhibited the highest number of complete failures at 17.8% (**O-4**). Furthermore, we see that short failures tend to affect a smaller fraction of active players, whereas longer failures tend to affect a larger part of the total active number of players (**O-5**).

5.3.1 Runescape

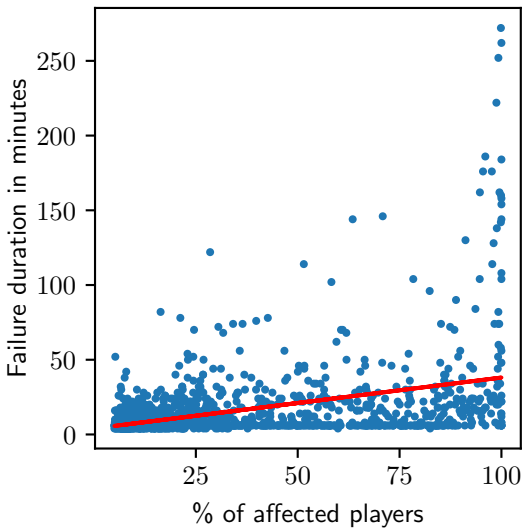

Figure 17: *Scatter plot showing the correlation between failure duration and % of affected players for Runescape.*

Runescape achieves parallellism through the use of instancing. This means that if one (or several) game servers responsible for separate instances fail, not all players are affected. Moreover, given sufficient capacity in the remaining instances, affected players can migrate to the working instances resulting in a quick recovery. In our data, failures such as these are characterized by affected player percentages significantly lower than 100%, and relatively short durations.

To distinguish complete failures from partial failures in our data, we define a complete failure as a failure where more than 95% of players are affected. We see that 201 out of 2341 (9.9%) failures can be seen as a complete failure, suggesting that the issue occurred in a critical part of the ecosystem. 50% of the failures affected less than 29% of players, further suggesting that while total failures do occur, most failures take place in non-critical components of the ecosystem, leaving other parts unaffected.

To verify whether failures that affect fewer players also last for a shorter time (and whether long failures affect more players), we look at the plot in Figure 17. The horizontal axis represents the percentage of affected players, and the vertical axis represents the failure

Service	Complete Failures	Total Num. Failures	% of complete failures
Hypixel	1	91	1.1%
Minehut	33	275	12.0%
Cubecraft	1	52	1.9%
The Hive	13	73	17.8%

Table 5: Complete failures of the four Minecraft services.

duration in minutes. Each point in the plot represents a single failure, and the red line is a linear regression line through the points. Firstly, the regression line shows that there is indeed a positive relationship between the failure duration and the percentage of affected players. In other words, short failures tend to affect fewer players, and longer failures tend to affect more players. To confirm this relationship, we perform a Spearman rank correlation test. This results in a correlation of 0.51, with a p-value of $2.24e{-}151$. This can be classified as a statistically significant strong correlation. Secondly, we see that the 12 longest failures all affected around 100% of the players. This shows that the outliers in terms of failure duration tend to affect all players.

5.3.2 Minecraft

Large-scale Minecraft services offer parallellism through instancing and zoning. This, as with Runescape, means that failures in particular resources do not have to result in complete failures. Indeed, this is observed in the failure statistics. Table 5 shows the number of complete failures for each Minecraft service using the definition of 'complete failure' from 5.3.1. We see that Hypixel and Cubecraft experience almost no complete failures, with 1.1% and 1.9% of failures being complete, respectively. Conversely, Minehut and The Hive do experience a fair amount of complete failures with 12% and 17.8% respectively, but the majority of failures are still partial. Between the four services, 50% of failures affect less than 23% of the players. This suggests that most failures occur in replicated resources, which do not affect all players upon failure. Failures where all players are affected at once appear to be rare.

To check the relation between failure durations and the fraction of players affected by those failures, we look at the plots in Figure 18. These are the scatter plots showing the percentage of affected players against the failure duration, with each point representing a single failure, and the red line being the linear regression line of these points. Hypixel and Cubecraft show a similar shape, different to that of Minehut and The Hive. The latter two show a strong concentration of failures around the 100% mark, whereas Hypixel and Cubecraft both only reach this point on a single occasion. Despite these differences, a positive relation between percentage of affected players and failure duration can be observed in all four services, as shown by the regression lines. To confirm this, we perform the Spearman rank correlation test for each of the services, giving the following results: Hypixel: 0.58, Minehut: 0.46, Cubecraft: 0.52, The Hive: 0.60. All of the results are statistically significant, and all but Minehut can be classified as strongly correlated. This supports the idea that short failures can be recovered from quickly, simply by the affected players moving to other zones or instances.

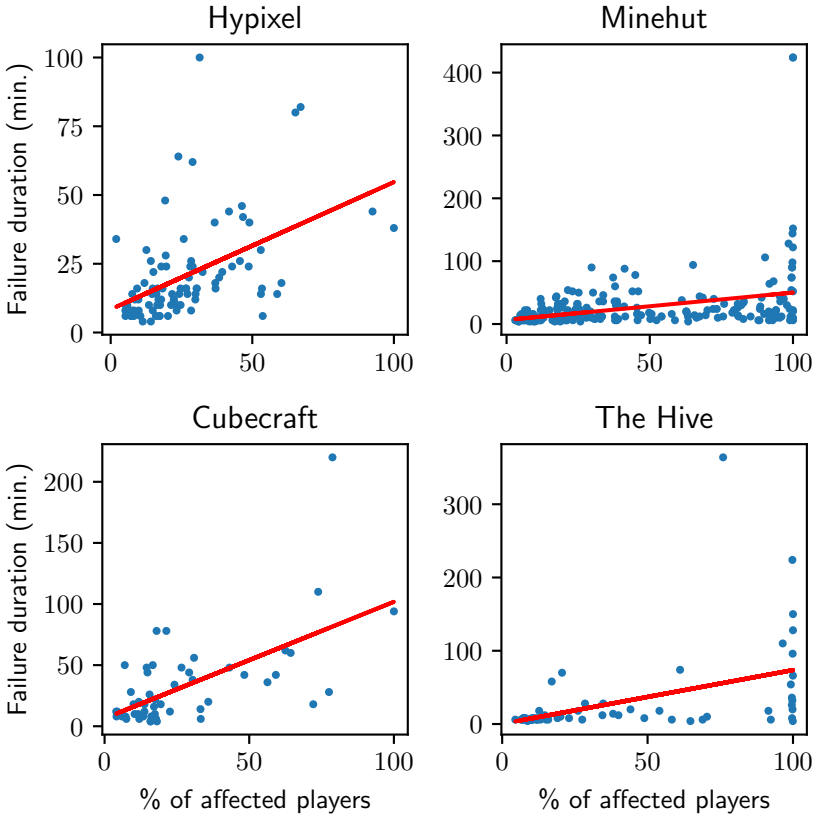

Figure 18: *Scatter plot showing the correlation between failure duration and % of affected players for Minecraft.*

5.4 The Consequences of Failures

Failures may have effects on the number of active players even after the failures have been resolved. To see if player counts are generally lower than normal after a failure has occurred, we compare the average player count in a time window before and after the failure, taken from the deseasonalized data (meaning that a value of 0 is precisely the player count that is expected at that moment in time). If the time window after the failure results in a lower average than the time window before the failure, we can conclude that the failure indeed resulted in a lower-than-normal active player count after the failure.

Figure 19 shows the percentage difference between the time window before the failure, and the time window after the failure. Negative values on the vertical axis mean that the post-failure windows had a lower average player count than the pre-failure windows. We see that all five games start with negative percentages, with only The Hive crossing over the zero line. This means that starting from 30 minute windows, The Hive actually gets higher average player counts after failures occurred. This stands in contrast to the other four games, which all stay in negative percentages, showing that player counts for these four games indeed are lower after failures, compared to before (**O-6**). Lastly, we see that all five lines trend upwards. This is expected, as longer windows mean that the average is being taken over a longer time frame, with the influence of the failure declining over time.

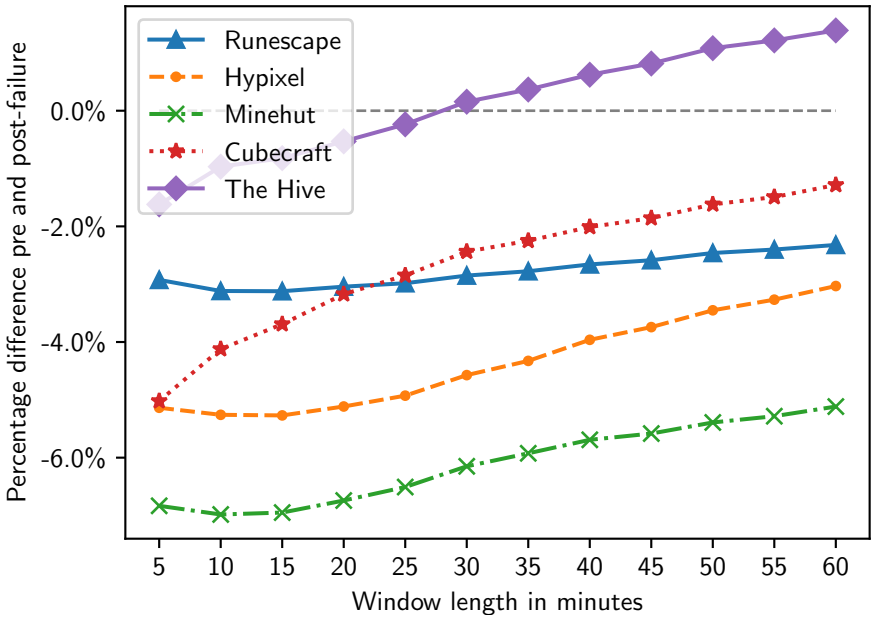

Figure 19: *Plot showing the percentage change in player count after failures, taken over different windows.*

5.5 Contribution of Planned Maintenance to Failures

As discussed in Section 5.2, the failure heatmap of Figure 14 suggests that a large part of failures in Runescape are caused by scheduled maintenance, as the highest failure concentration occurs around the time where the least amount of players are active. To investigate this, we use a list of all officially announced game updates [13], sourced from the official Runescape website. This list contains the dates of each update. Seeing as the records do not contain precise times, we assume that failures that occur on dates with known updates are a direct result of those updates. This allows us to separate outages related to scheduled updates from unexpected failures. We find that 34.8% of failures in Runescape can be attributed to planned maintenance, which reveals a significant point of improvement (**O-7**).

Figure 20 shows two heatmaps. The upper heatmap shows the failure counts including updates, and the lower heatmap shows failure counts with the updates filtered out. We see that after excluding updates, the heatmap becomes more uniform. The strong concentrations of failures on Monday, Tuesday, and Wednesday afternoon are diminished. This supports the idea that the high concentration of failures observed around times of low traffic can be attributed to scheduled maintenance. Interestingly, only Monday, Tuesday, and Wednesday appear to be affected by this. The profile of Thursdays shows the same characteristics across both heatmaps. This, combined with the fact that Thursday still shows a high concentration of failures at a time with few players, suggests that this hotspot is still caused by maintenance. However, this maintenance is not officially announced as being related to updates.

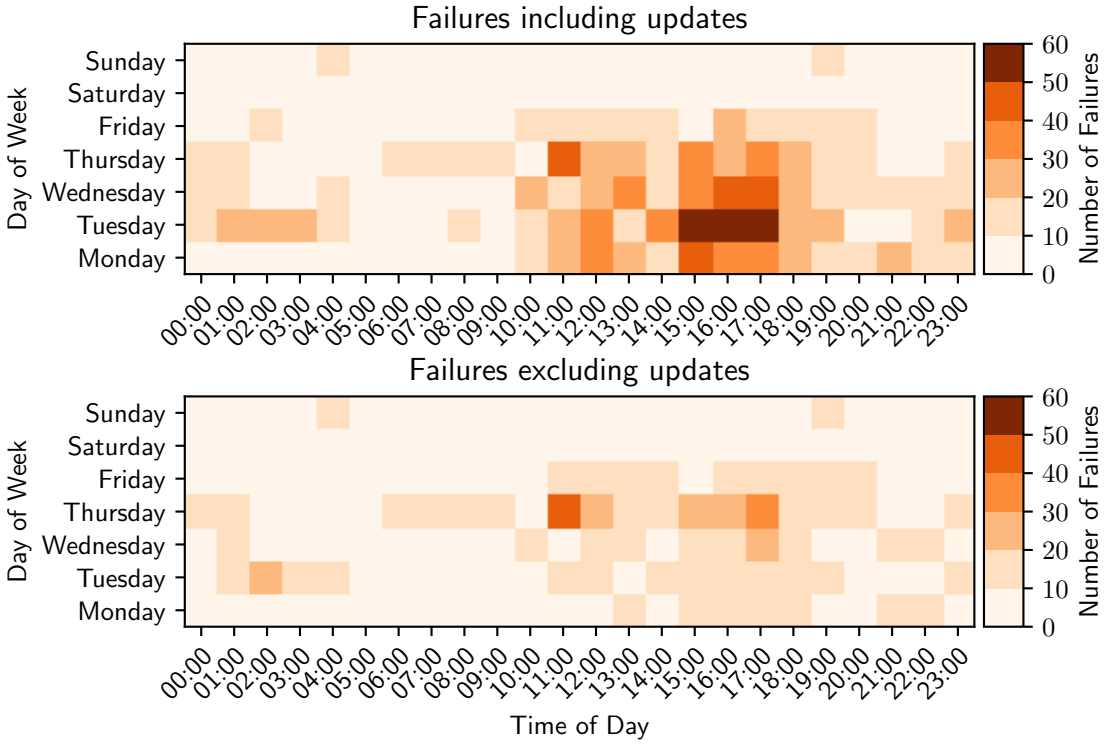

Figure 20: *Failure density heatmaps for Runescape, including and excluding known updates.*

In terms of failure characteristics, we observe some change in the statistics when comparing the original set of failures to the set with updates filtered out. The average duration of the failures drops from 21 minutes to 15 minutes, suggesting that the outages caused by scheduled maintenance tend to last longer than other outages. The average interarrival time grows from 1 day to 2 days, and the average percentage of affected players drops from 41.7% to 33.3%, indicating that scheduled maintenance tends to affect more players than other types of failures do. Lastly, comparing the total number of failures, we see that 778 of the 2235 total failures (34.8%) can be attributed to scheduled maintenance. This, combined with the fact that the failures attributed to scheduled maintenance tend to be more severe than other failures, suggests that a significant improvement in availability could be made by implementing rolling updates; a way to deploy updates without downtime. This is a feature commonly seen in cloud computing, but less common in MMOGs.

5.6 Comparison Between Games

This subsection discusses the similarities and differences in failure characteristics between the analyzed games. We find significant similarities between the analyzed games in failure characteristics and traffic shape. Differences are seen in their active player bases and temporal failure patterns (**O-8**).

Comparing Figure 14 and Figure 16 reveals that the four Minecraft services, and Runescape, share a very similar traffic shape. This is notable for several reasons. Firstly, the data for

	Hypixel	Minehut	Cubecraft	The Hive	Runescape	RS w/o updates
Num. Failures	91	275	52	73	2341	1457
Avg. Duration	20m	24m	32m	28m	21m	15m
Avg. Interarr.	3 days	1 day	5 days	4 days	1 day	2 days
Avg. % affected	26.3%	40.6%	27.3%	37.2%	41.7%	33.3%

Table 6: Summary of failure statistics for all analyzed games.

Runescape spans a much larger time frame, going back to 2007, compared to the data for Minecraft which starts in 2020. Secondly, Runescape and Minecraft are quite dissimilar games, with potentially different target audiences. These factors do not appear to have an effect on the traffic shape.

Table 6 shows the summary of the failure statistics for Minecraft and Runescape. Comparing these failure statistics also shows a strong similarity between the Minecraft services and Runescape. For instance, the average interarrival time and the average percentage of affected players of Runescape matches that of Minehut, and Runescape's average failure duration matches that of Hypixel, despite the mentioned differences. Furthermore, the percentage of complete failures of Runescape also falls within the range of the Minecraft services. For Runescape with updates filtered out, we see similar results. The average interarrival time and average percentage of affected players fall within the range set by the other games, with the average failure duration being 25% lower than the lowest average failure duration in the other games. These statistics suggest that failures in MMOGs follow similar patterns across the board, and are not strongly influenced by the specifics of each game.

There is a substantial amount of variation between the 'player base' (i.e. the average number of online players on an average day) of each of the analyzed sources. The four Minecraft services range from 12,002 to 86,976, and Runescape has an average player count of 99,051. Given this variation, a related variation in failure statistics could be expected. However, no such difference can be observed. There appears to be no link between the average number of players within the range of 12,000 to 100,000, and the failures that occur.

Further differences can be observed in the temporal pattern of failures. The four Minecraft services share some similarity in when most failures occur, while Runescape exhibits a completely different pattern. This is notable, as the general failure statistics do not differ strongly between the games.

6 Threats to Validity

This section discusses the main threats to the validity of this work.

Firstly, despite having built a functional prototype of the game failure analysis data pipeline, we do not have a real-world large scale deployment of the system to validate its working in such scenarios. A large scale deployment may reveal unforeseen issues or flaws in the architecture. Additionally, use of the system for a wide variety of diverse games may potentially reveal scenarios where failure analysis would benefit from storing data which does not have a place in the current format. While the majority of the pipeline is not dependant on the final format of the saved data (the data lake does not impose format restrictions), the game failure storage format may require revisions which could negatively affect tools built around it.

Secondly, the method used to detect failures from player counts is limited by what is clearly visible in the time series graph by a human observer. This is due to the fact that the performance of the detection algorithm is measured against a manually labeled data set, and the expected visual shape of the peaks is integral to the 'peak expansion' part of the method. There may be failures present in the data that do not manifest in clearly recognizable peaks. Therefore, even if the automatic detection was 100% in line with manual labeling, some failures may still not be recognized by the algorithm.

Ideally, we would have a ground truth with which we could compare the results from our gathered data. This would allow us to better validate the detection of failures, being able to take into account the aforementioned limitations. However, the reality of working with real MMOGs outside our control prevents such validation. The use of self-reported player counts may therefore introduce inaccuracies. For instance, the online player count could be a rolling average, or the rate at which the reported player count is updated may not be constant. This could result in the sampled values not accurately representing the state of the system. For our results, we therefore make the assumptions that all failures resulting in outages are reflected in the player counts, MMOG operators do not manipulate self-reported player counts, and that they accurately depict the real state of the system. However, the failure characteristics found in this thesis are in line with characteristics found in related studies [26], so these assumptions appear to be fair.

Lastly, the low number of analyzed games is a limitation to the generalizability of the results. Analyzing a larger number of games may reveal differences or similarities that were not found in this work, and would allow for more general statements on the nature of failures in MMOGs.

7 Conclusion and Future Work

MMOGs have become a large industry, with demanding user bases and increasingly complex underlying systems. This complexity makes failure almost inevitable, so it is important to gain an understanding on the ways these systems fail. To this end, we have posed three research questions, which we have addressed in Sections 3, 4, and 5.

RQ1: How to collect availability data from game services?
We have presented a system architecture that manages the collection, storage, and analysis of online game availability data, and a unified storage format for said data. Additionally, we have implemented a functional prototype to validate the proposed system.

RQ2: How to detect game failures using publicly available data?
We have shown a method for the automatic detection of failures from high-frequency online player count time series by observing changes over time, allowing for the failure analysis of large datasets where manual labeling is infeasible.

RQ3: What are the characteristics and statistical properties of online game service failures?
We have performed failure analysis of two highly popular MMOGs (Runescape and Minecraft). This has shown that failures in MMOGs occur frequently, with none of the analyzed games achieving an average interarrival time of over 5 days. Failures tended to be short, with average durations under half an hour. Interestingly, most failures do not occur during the most busy periods. The analyzed games exhibited similar failure characteristics despite differences in game style, player base, and analyzed time periods. Lastly, planned maintenance was identified to be a big contributor to MMOG outages, with up to 35% of detected failures for Runescape being the result of planned updates.

In this work we have focused on tackling the main challenge present in the failure analysis of MMOGs, namely the lack of available data. We have shown that failure data can be extracted from very limited sources, and have provided the design for a system that allows anyone to collect, analyze, and share this data. We hope this work can facilitate the analysis of a wider range of games, enabling more comprehensive comparisons and further expanding the understanding of MMOG failures.

References

[1] `https://mmo-population.com/top/2021`. [Accessed 10-Sept-2021].

[2] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004.

[3] D. Baszucki. `https://blog.roblox.com/2021/10/update-recent-service-outage/`, 2021. [Accessed 3-Nov-2021].

[4] C. Chambers, W.-c. Feng, S. Sahu, D. Saha, and D. Brandt. Characterizing online games. *Networking, IEEE/ACM Transactions on*, 18:899 – 910, 07 2010.

[5] K.-T. Chen, P. Huang, G.-S. Wang, C.-Y. Huang, and C.-L. Lei. On the sensitivity of online game playing time to network qos. 04 2006.

[6] J. Cohen. A coefficient of agreement for nominal scales. *Educational and Psychological Measurement*, 20(1):37–46, 1960.

[7] EVE Development Team. `https://www.eveonline.com/news/view/the-second-timer-in-m2-xfe`, 2021. [Accessed 24-Apr-2021].

[8] P. Garraghan, P. Townend, and J. Xu. An empirical failure-analysis of a large-scale cloud computing environment. In *2014 IEEE 15th International Symposium on High-Assurance Systems Engineering*, pages 113–120, 2014.

[9] Y. Guo and A. Iosup. The game trace archive. In *NETGAMES*, pages 1–6, 11 2012.

[10] C. Hall. `https://www.polygon.com/2021/1/5/22214982/eve-online-world-record-massacre-m2-xfe-ghost-titans`, 2021. [Accessed 24-Apr-2021].

[11] Hypixel. `https://status.hypixel.net/history`. [Accessed 23-Sept-2021].

[12] Jagex. `https://www.runescape.com/community`. [Accessed 08-Sept-2021].

[13] Jagex. `https://runescape.wiki/w/Game_updates`. [Accessed 12-May-2021].

[14] B. Javadi, D. Kondo, A. Iosup, and D. Epema. The failure trace archive: Enabling the comparison of failure measurements and models of distributed systems. *Journal of Parallel and Distributed Computing*, 73(8):1208–1223, 2013.

[15] D. Kondo, G. Fedak, F. Cappello, A. Chien, and H. Casanova. Resource availability in enterprise desktop grids. *Future Generation Comp. Syst.*, 23:888–903, 08 2007.

[16] N. Krecklow. `https://www.minetrack.me/`. [Accessed 02-Jun-2021].

[17] Y.-T. Lee, K.-T. Chen, Y.-M. Cheng, and C.-L. Lei. World of warcraft avatar history dataset. In *Proceedings of the Second Annual ACM Conference on Multimedia Systems*, MMSys '11, page 123–128, New York, NY, USA, 2011. Association for Computing Machinery.

[18] C. Marshall. `https://www.polygon.com/22386085/minecraft-realms-down-outage-mojang-support`, 2021. [Accessed 3-Nov-2021].

[19] S. Musil. `https://www.cnet.com/tech/gaming/roblox-games-back-online-after-3-day-outage/`, 2021. [Accessed 3-Nov-2021].

[20] V. Nae, L. Köpfle, R. Prodan, and A. Iosup. Autonomous massively multiplayer online game operation on unreliable resources. In *Proceedings of the International C* Conference on Computer Science Software Engineering, C3S2E13, Porto, Portugal - July 10 - 12, 2013*, pages 95–103, 2013. 6th International C* Conference on Computer Science and Software Engineering, C3S2E 2013 ; Conference date: 10-07-2013 Through 12-07-2013.

[21] C. Robert, C. William, and T. Irma. Stl: A seasonal-trend decomposition procedure based on loess. *Journal of official statistics*, 6(1):3–73, 1990.

[22] RuneScape Wiki. `https://runescape.fandom.com/wiki/Wilderness_and_Free_Trade_Vote`. [Accessed 22-Apr-2021].

[23] RuneScape Wiki. `https://runescape.fandom.com/wiki/ClusterFlutterer`. [Accessed 22-Apr-2021].

[24] P. Rykała. The growth of the gaming industry in the context of creative industries. *Biblioteka Regionalisty*, (20):124–136, 2020.

[25] B. Schroeder and G. A. Gibson. A large-scale study of failures in high-performance computing systems. *IEEE Transactions on Dependable and Secure Computing*, 7(4):337–350, 2010.

[26] S. Talluri, L. Overweel, L. Versluis, A. Trivedi, and A. Iosup. Empirical characterization of user reports on cloud failures. In *Proceedings of the 2nd IEEE International Conference on Autonomic Computing and Self-Organizing Systems*, ACSOS '21.