Vrije Universiteit Amsterdam

Universiteit van Amsterdam

Master Thesis

ProcGen: Designing a Serverless Procedural Content Generation System for Virtual Environments

Author: Misha Rigot (2639323)

1st supervisor:	Prof. dr. ir. Alexendru Iosup
daily supervisor:	Ir. Jesse Donkervliet
2nd reader:	Dr. ir. Daniele Bonetta

A thesis submitted in fulfillment of the requirements for the joint UvA-VU Master of Science degree in Computer Science

August 25, 2025

"To attain knowledge, add things every day. To attain wisdom, remove things every day."
by Laozi

Abstract

The game industry is one of the largest entertainment sectors, yet game content creation remains a time-consuming and expensive process. Procedural Content Generation (PCG), which algorithmically generates game content, offers a solution for reducing development costs while increasing replayability and content variety. This thesis investigates how serverless computing can further enhance PCG by improving the scalability, cost-efficiency, and flexibility of PCG systems. This work is presents several challenges. Designing a PCG system for serverless deployment requires careful consideration of the unique characteristics of serverless computing, such as cold-start delays, execution time limits, and statelessness. Moreover, game development introduces additional constraints such as the need for low latency response, high content quality and diversity. These requirements vary widely between game genres. Adopting a client-server architecture for PCG also introduces overhead from network communication, particularly in terms of payload size and serialization costs. Addressing these challenges while maintaining system modularity and scalability requires a well thought out design approach. This research presents a novel PCG system divided into two sub-systems: one that generates content in an abstract format within a serverless environment, and another that transforms this abstract content into concrete game elements. This design capitalizes on the benefits of serverless architectures, such as dynamic scaling and reduced operational overhead. A prototype of the system is implemented in the Unity game engine, and is able to generate content both serverlessly, as fully locally to improve adoption. The prototype is able to generate 2-dimensional dungeon layouts. The system's performance is evaluated through a set of experiments assessing scalability, comparing content retrieval policies, and measuring the latency overhead introduced by serverless deployment. Findings demonstrate that serverless computing allows for scalable content generation, albeit with trade-offs in terms of response time variability. Additionally, we find that content retrieval policies can have a large impact on latency, fitness score and

duplicate rate, where each policy provides a different trade-off. These findings provide valuable insights for optimizing PCG systems in serverless environments, offering practical recommendations for game developers.

Contents

List of Figures

LIST OF FIGURES

List of Tables

LIST OF TABLES

Chapter 1

Introduction

Video games are a popular form of entertainment, with the worldwide number of players being roughly 3.38 billion in 2023, and with revenues generated globally in the games industry reaching $184 billion (1). In contrast, in 2023 global streaming revenue is estimated at a gross revenue of $65.7 billion (2) and the recorded music industry an estimated gross revenue of $28.6 billion (3).

For a video game to be a success, it needs to have content such as a story, an environment, 2-D or 3-D models, audio, etc. The type and amount of content required differs per game, and is dependent on factors such as game genre, scope, target audience, target platform, etc. Content is often costly to make, as it typically requires a significant amount of work hours to produce. For a content-heavy game, the amount of time required can go up into the thousands to hundred thousands, depending on the scope.

Procedural content generation (PCG) is the generation of content via an algorithm. PCG can alleviate game developers of (a part of) manual content creation. This can save developers significant amount of time and cost, which allows them to focus on other parts of game development. Another benefit of PCG is increasing the variation and replayability of games, as content can easily be regenerated to produce different content than a previous playthrough, keeping the player engaged. Additionally, PCG can generate vast amounts of content quickly, enabling the creation of expansive worlds without a proportional increase in resources. Furthermore, PCG provides storage benefits, as content can be generated on the fly, making the game require less assets to be installed upfront such as entire levels, resulting in smaller installation sizes. Examples of games making heavy use of PCG are Minecraft, which generates entire 3-D worlds with different biomes, Terraria generating 2-D worlds, Spelunky generating platformer levels, and the Diablo franchise, generating dungeons.

Beyond games, PCG can also benefit the metaverse, a persistent, immersive virtual shared environment (4, 5, 6). Like video games, the metaverse requires vast amounts of content, including 3-D environments and interactive elements. However, due to the metaverse's scale and real-time adaptive nature, manual content creation can often be impractical. As the metaverse continues to grow, PCG will play a large role into shaping the metaverse, to offer a scalable solution to diverse content generation, ensuring its long-term sustainability.

1.1 Problem Statement

Procedural content generation has been studied extensively in academic contexts (7, 8, 9, 10, 11), as well as used in the industry (12, 13, 14, 15, 16) as a method of generating content to increase replayability and content variety, and to reduce manual development efforts required for content creation. However, the majority of PCG implementations described in academic literature and in commercial applications generate content locally from within the game engine, or use traditional client-server architectures for remote content generation (such as dedicated servers, VMs, or containerization).

In contrast to these traditional deployment approaches, serverless computing has emerged in recent years as a new cloud computing paradigm that offers on-demand scalability at reduced operational effort with fine-grained billing. Serverless has seen an increase in popularity for web-services, data processing pipelines, and has seen some use in game-related back ends (e.g., leaderboards, analytics or matchmaking). However, the application of serverless in the context of PCG remains largely unexplored (17, 18). To the best of our knowledge, no general purpose, open-source, research-focused serverless PCG system exists to help guide both developers and researchers in leveraging serverless for content generation in games.

The lack of such a system raises several problems. First, it is unclear how to design a serverless PCG architecture that adheres to requirements common in PCG systems in games, within the constraints posed by the serverless architecture. Second, it is unknown how such a design can be integrated into existing game engines. Third, no experimental data exists on how serverless deployments affect the performance of PCG systems in terms of scalability, fitness score and duplication rate.

1.2 Research Questions

Our objective is to create a serverless PCG system to investigate opportunities of serverless in PCG. To this end, we have constructed the following research questions:

RQ-1 How to design a serverless procedural content generation system? To construct a serverless PCG system, we first have to know what design fits our requirements. These requirements first have to be established, based on how PCG is applied in a non-serverless setting. Designing such a system is challenging, as no existing serverless PCG systems exist to the best of our knowledge. In addition, our design should be fitting for arbitrary content types and methods. This property increase complexity as we have to design with generality in mind.

RQ-2 How to implement a serverless procedural content generation system? It is important to implement our design of a serverless PCG system, as it contributes to the validation of our design, as well as demonstrating its potential. Implementing this system is challenging, because serverless is not commonly applied in gaming, specifically in Unity, as a core part of the game's content generation process. This could present us with technical challenges in order to fit serverless into the existing Unity ecosystem.

RQ-3 How to evaluate and validate a serverless procedural content generation system? Evaluating and validating the system is important, as it allows us to verify that the requirements we set in our design are met, by both the design and the implementation of the system, in addition to demonstrating the performance of our system. Evaluating and validating a serverless PCG system presents several unique challenges. First, serverless architectures introduce performance variability due to factors such as cold starts and unpredictability in resource allocation. Second, isolating the impact of serverless deployment requires careful experimental setup, as it requires distinguishing between delays introduced by the serverless infrastructure versus the PCG algorithm itself.

1.3 Research Methodology

To answer each research question, we use the following approaches. Each approach's number corresponds to the RQ number in the previous subsection.

We answer each of our research questions by applying the AtLarge design process (19). This design process is divided into several steps, called the *Iterative Basic Design Cycle*. Each research question is answered by following a subset of the steps.

RM-1 We answer RQ1 by first conducting a requirements analysis through examining existing literature on PCG, through the insights from our proposed reference architecture, and through domain expertise. The requirements are then iterated upon through knowledge gained during both the design and implementation of the system.

RM-2 To understand how to implement the design, we implement a prototype of the system following an iterative and incremental development process. This is in line with the AtLarge design process, corresponding to step *Implementation to Analyze the Design*. We essentially co-evolve the design and implementation, i.e., adjusting the design based on constraints discovered through implementation. Furthermore, we adopted a separation-of-concerns approach by decoupling abstract content generation from concrete instantiation, supporting extensibility and the integration into different engines. To choose a content type that the procedural generator in our prototype generates, we explore commonly generated content types by surveying the literature on PCG.

RM-3 We perform an experimental analysis on the prototype developed to answer RQ2. We do this through a performance evaluation on several key metrics, such as response time, throughput, fitness score, and duplicate rate. These metrics are measured in different system configurations to gain insight into how different configurations influence these metrics. Additionally, we validate our system through these experiments, to test whether our system meets the requirements as defined in our design, the answer to RQ1.

1.4 Research Contributions

This thesis makes the following contributions.

RC-1 (Conceptual) A design of a serverless procedural content generation system for games, that is split into an abstract content generation system and a concrete content generation system, allowing for a modular, extensible and scalable system.

RC-2 (Technical) A prototype of a serverless procedural content generation system for games. We publish our prototype as a free and open-source software (FOSS) and make it available on GitHub (20)

RC-3 (Technical) An open-source Unity package of a procedural content generation system. This allows developers to use the system in their Unity projects. The package supports parameterized generation requests.

RC-4 (Experimental) Real-world evaluation of the implemented system. We conducted a set of experiments on the deployed version of the prototype in AWS to test scalability, latency, content diversity, and fitness. We explore trade-offs in content retrieval policies. The results provide evidence that serverless can be a viable architecture for running procedural content generation systems at scale.

RC-5 (Technical) A system that can be used as a reference for using serverless in Unity. By integrating serverless in a Unity package via Infrastructure as Code, our work serves as a concrete example for developers interested in offloading compute-intensive or content generation tasks from the Unity client to a serverless infrastructure.

1.5 Thesis Structure

This thesis is structured as follows. In Chapter 2 we provide background on both PCG and serverless computing to provide the reader with additional context. Chapter 3 analyses the requirements and describes the design of our serverless PCG system based on these requirements. Chapter 4 describes how we implemented the design of the previous chapter as a prototype serverless PCG system as a Unity package. We evaluate this system in Chapter 5, by running simulations and benchmarks on the system in various configurations. In Chapter 7 we summarize our work and outline future work opportunities.

Chapter 2

Background

This chapter provides context required to understand the concepts covered in this thesis. The first being procedural content generation, covered in the first section, and the second being serverless computing, which is covered in the second section.

2.1 Procedural Content Generation

This section explains what PCG is, and provides a summary of both the scientific literature on PCG and its use in games.

2.1.1 What is PCG?

Procedural content generation is the generation of content procedurally through a system that employs an algorithm that generates the content.

Manual content creation is time consuming. PCG allows developers to save time on manual content creation. It can generate potentially infinite amounts of variety and novelty. In the context of games, PCG improves replayability of the game, as content differs between each playthrough. This can also make games more dynamic, and facilitate emergent gameplay, as players need to adapt to the game's generated content. Memorizing a level is, in the case of level generation, not an option. Replayability is a desiring characteristic of games, as it keeps players engaged with a game over longer periods of time.

One common pitfall of PCG is the *10,000 bowls of oatmeal problem* coined by Kate Compton in her GDC talk about PCG (21). The meaning of this problem is the idea that, while a procedural generator can generate truly unique content, the content may not be perceived as such, similar to how different bowls of oatmeal are also technically unique, yet indistinguishable in practice. Mitigating this phenomenon requires generators

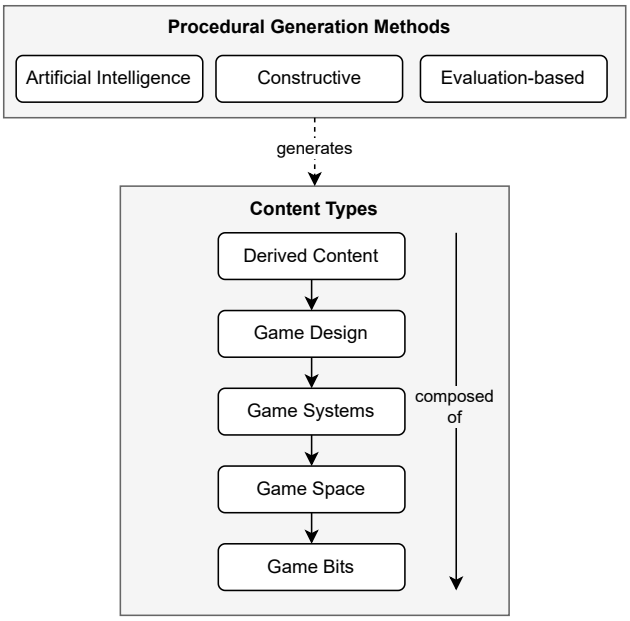


Figure 2.1: Procedural Content Generation methods and Content Types.

that produce content with high diversity, ensuring not only technical variation but also perceptual distinctiveness from the player's perspective.

2.1.2 Types of content

Several types of content can be produced by PCG systems, as outlined by (7), where the authors present six different types that content can fall in. These content types are depicted in the lower part of 2.1. The content types are Game Bits, Game Space, Game Systems, Game Design, and Derived Content, in order of lower- to higher-level. Each type has multiple sub-types, for instance, the type Game Space has the sub-types Indoor Maps, Outdoor Maps, and Bodies of Water. Each content belonging to a type may be built from content of types of a lower level. For example, a Game Space might consist of a forest (Game Space: Outdoor Map), that contains several maze-like dungeons (Game Space: Indoor Maps). The forest contains tree meshes (Game Bits: Vegetation) that might also be generated, along with their generated textures (Game Bits: Textures), and so on.

2.1.3 Types of generators

Procedural content generators employ a variety of generation methods that generate the content. In our earlier work (22), we categorized these generation methods into three types, namely Artificial Intelligence, Constructive, and Evaluation-based. These generation methods are depicted in the upper part of 2.1. Each category has multiple methods. Artificial Intelligence-based methods use machine learning algorithms, that are trained on data with existing content, to generate new content. Constructive methods run in fixed-time and create content by applying predefined rules, constraints, and procedural algorithms. Evaluation-based methods evaluate their content after generation, and in the majority of cases, search for better-evaluating content over multiple iterations. These methods thus commonly use a search algorithm to iteratively search the space of content and an evaluation function to evaluate each content item until content is generated that meets the set criteria (or a threshold on the maximum number of iterations is reached).

2.1.4 Related work

PCG has been researched through the creation of PCG systems, and through meta-research, e.g., through surveys.

In terms of surveys, the authors of (7) present a taxonomy of both content types and generation methods. For each content type, they discuss systems that generate that content type and which generation methods they use. Other surveys focus on specific content types, such as (8)'s work specifically focusing on the procedural generation of dungeons, (23) focusing on the generation of buildings, or (24) focusing on puzzle generation. In contrast, other surveys focus on a specific generation method, such as search-based PCG (11), deep learning based methods (25), or through quality diversity (26).

Other research typically present PCG systems using different generation methods. For example, such as through prompt engineering (27), evolving *Super Mario Bros* levels using grammatical evolution (28), or a quest generator using a planning approach (29). These systems typically attempt to improve the quality of the content generated.

2.1.5 PCG in the industry

PCG is commonly used in the gaming industry by both small and AAA companies. Well known examples of games that are using PCG for world generation for 3D worlds are Minecraft, No Man's Sky and Valheim, and Terraria for 2D worlds. Levels, e.g., dungeons, are also commonly generated, for example in the Diablo series, Spelunky, and more recently

Lethal Company. Other games include Borderlands, which generates weapons, and Dwarf Fortress, generating entire worlds with scenarios.

Game engines also employ PCG as tools that can be used by the game developers using the engines. Both Unity and Unreal Engine have tools for creating and (de)forming terrain. Unreal Engine has a first-party plugin, the Procedural Content Generation Framework[1], that allows for the procedural placement of meshes by setting up a node graph that contains the logic for which meshes to spawn at which locations on another static mesh such as terrain. Both engines have plugins available on their marketplaces that provide PCG tools for other game content, such as terrain and indoor spaces.

2.2 Serverless

This section explains serverless by first describing operational challenges in software development deployment, after which an explanation of serverless is given, and how it addresses these challenges. Finally, we discuss how serverless and games can or are combined an practice.

2.2.1 Operational Challenges in Software Development Deployment

There are several operational challenges in software development deployment. These operation challenges can be time consuming and therefore costly.

One of such challenges is maintenance, being an integral part of software development. Software is no longer only provided as data on disk. Instead, software is more often than not running as a web service, or running on a computer connected to the internet. This makes updating existing software important, and thus an important challenge is to push updates with as little downtime as possible.

Another operational concern for any deployed application is its reliability and availability. Reliability means the ability for a system to fulfill its tasks without producing errors or showing performance degradation. In addition, availability means the percentage of time the service is up and running.

Another primary concern for operations is the cost of running the infrastructure. For non-serverless cloud services such as IaaS, the user typically keeps the deployed application running continuously, even when the system is idle. This has cost implications.

[1] https://dev.epicgames.com/documentation/en-us/unreal-engine/procedural-content-generation–framework-in-unreal-engine

Thus, operations can require considerable time and effort and minimizing this requirement allows a software organization to focus more on the core business and application logic. Reducing operations tasks enables organizations to invest less in staff that is specialized in operations, and instead allows for investing in more software developers that help develop the core business.

Next to operational needs, deployed applications often have varying scalability needs, workloads and request patterns. A typical example of varying request patterns is users engaging with the system at varying times during the day or week. For instance, for a business application, users will typically engage with the system during work hours on weekdays, for instance. Another example is in the context of video games, where a large amount of users will play the game on initial launch of the game, or when a new update has been released. Then, gradually, the users will decrease over time. But even then, user engagement will still follow a pattern during day/night cycles or at certain days in the week. Ideally, the application's infrastructure would dynamically scale up and down, based on the demand, or workload, on the system. This means scaling the infrastructure up when there is more load on the system, and down when that load decreases. This is challenging, as a balance needs to exist that attempts to match the available infrastructure resources to the user demands in such a way that maximizes both performance and cost efficiency. Having too many resources, compared to the current workload is less cost efficient, and having less resources than needed for the current workload is detrimental for performance.

Another concern is time-to-market, as additional required operational efforts to deploy an application can cost the developer valuable time, which reduces the time-to-market and therefore has a negative impact on the developer's business.

2.2.2 What is serverless?

In (30), the authors propose a refined definition of serverless:

> Serverless computing is a cloud computing paradigm encompassing a class of cloud computing platforms that allow one to develop, deploy, and run applications (or components thereof) in the cloud without allocating and managing virtualized servers and resources or being concerned about other operational aspects. The responsibility for operational aspects, such as fault tolerance or the elastic scaling of computing, storage, and communication resources to match varying application demands, is offloaded to the cloud provider. Providers apply utilization-based billing: they charge cloud users with fine granularity, in

> proportion to the resources that applications actually consume from the cloud infrastructure, such as computing time, memory, and storage space

The benefits of serverless include minimal operational efforts from the user, as this is the responsibility of the cloud provider. Additionally, due to fine-grained billing, running an application in a serverless environment can potentially be very cost-efficient. Furthermore, in the majority of cases, serverless platforms are elastic, meaning they scale automatically depending on increased/decreased demand. This scaling can be nearly infinite.

According to the study of (30) on the definition of serverless and its characteristics, serverless is considered to consist of multiple services, including FaaS and BaaS. Other *X-as-a-Service* service models, such as CaaS, PaaS, SaaS and IaaS are not considered serverless. Examples of FaaS on popular cloud platforms are AWS Lambda on Amazon Web Services (AWS), Cloud Functions on Google Cloud Platform (GCP), and Azure Functions on Microsoft Azure. Examples of services that are considered BaaS are AWS RDS (relational database), Azure Blob Storage (object storage), Google Cloud Datastore (NoSQL datastore), and AWS SQS (message queue). An application can, thus, make use of a combination of FaaS and BaaS services in order to be considered as employing serverless architecture.

Certain types of applications or services are more fitting to be run in current serverless environments than others. Firstly, applications that have components or functions that are stateless, as each serverless function is stateless as well. Statelessness means that each call to a serverless function is independent of other serverless function calls. Individual serverless function calls have no knowledge of or reliance on each other. Secondly, serverless functions should require a relatively small amount of memory and have a relatively short runtime. Short runtime requirements are present, because FaaS offerings have a maximum runtime, e.g., 15 minutes in the case of AWS Lambda. Additionally, applications should be able to allow for variation in response time, as slow-starts can increase the response time of the application. The aforementioned characteristics of serverless functions are not set in stone and could potentially be less strict in future serverless environments (30, 31, 32) .

Serverless also has limitations. One of such limitations is cold start times, which is the time it takes for a serverless function to initialize. These cold start times are present when there are no existing available serverless function containers running for the respective service, in which case it needs to boot up a fresh container, which takes more time than running on an already initialized container.

Another limitation is the often limited run duration of a serverless function. For example, the most popular serverless function service, AWS Lambda, has a maximum run duration of 15 minutes. These durations vary across different serverless providers. For example, Google Cloud Functions have a timeout of either 9 minutes or 60 minutes, depending on whether the first or second generation of the service is used.

2.2.3 Serverless and Games

Games often have different performance requirements compared to traditional software. These requirements are often driven by the real–time and interactive nature of games, where consistency and responsiveness are critical. Traditional software often have less strict latency requirements, as opposed to games, which demand low latency, consistent frame rates, and minimal response variability to ensure an immersive and smooth user experience.

Game performance can be affected by numerous bottlenecks, such as rendering workloads that stress the graphical processing unit (GPU), game logic computation that require work from the central processing unit (CPU), high memory requirements for loading entire game worlds in memory, or high frequency data exchanges with low latency requirements. Networked games also introduce requirements on latency, jitter, stability, tick-rate or throughput. As a result, performance degradation can have a large impact on player experience and the success of a game.

Online games primarily utilize two networking models: client-server and peer-to-peer. In the client-server model, a set of clients connect to a single server. This server, most often authoritative, often processes state changes and sends these state changes back to clients. This model offers more security and control. In the peer-to-peer model, each client directly connects to all other clients in the game. The benefit is not having to provide servers to players, as the players themselves are the servers. However, this has security implications, as there is no central authority. Additionally, inconsistency in game state can occur due to the lack of a central authority. Both models often maintain the game's state in-memory and maintain longer-running game sessions, and thus are not easily translated into a serverless implementation.

However, not all types of games or features in games require continuous state synchronization and low latency, lending themselves well for running in a serverless environment. These features include leaderboards, achievements, matchmaking, authentication, authorization, and slower-paced games such as turn-based games. While state can most certainly be a part of these features and genres, the state can more easily be managed inside a database,

whereas typical game state updates that need to be synchronized very frequently with clients are more suited to store the state in-memory.

Furthermore, some procedural generation tasks may also be decoupled from the main game loop. When generation does not need to happen in real-time, or with lower latency requirements, these tasks can be offloaded to a serverless system, freeing up resources on the client side and improving performance and thus player experience.

Finally, recent research has begun to explore the feasibility of serverless architectures in more complex game scenarios. Donkervliet et al. (18) envision large-scale MVEs to be architected as serverless systems. In (17), serverless is used as a backend architecture to increase the scalability of MVEs, which increased the supported player count by 140. These studies show a growing research interest in combining serverless with games, demonstrating its potential of selectively offloading the right game systems to improve scalability and performance.

Chapter 3

ProcGen: Design of a Serverless Procedural Content Generation System for Virtual Environments

In this section, we describe the design of the system, which answers research question 1: *How to design a serverless procedural content generation system?* We follow the AtLarge framework for design (19). This framework provides design principles to aid in the design of distributed systems and services.

3.1 Stakeholders

We identify three stakeholders. These stakeholders drive the requirements created for the design of the system.

ST-1 **Game developers** want to use PCG to enrich their games with content. Game developers might want to generate as much content as possible, or specific parts based on the game being developed. They want to have as much control as possible with as little amount of complexity as possible, allowing them to generate content that fits their game design ideas, while still saving development time.

ST-2 **PCG scientists** who wish to generate content to be used for PCG related experiments. They want an easily extendable system that has the potential for a large variety of generation methods. Additionally, they have a need for modularity in content retrieval behavior, to easily test different configurations of the system.

ST-3 Scientists from a non-PCG field that require a virtual world for their experiments. These stakeholders benefit from generation that can be easily controlled through a user-interface, as opposed to through code.

ST-4 Players are an indirect user of the system, as they play the game or application that contains the content generated by ProcGen. They require acceptable latency to improve their quality of experience, as well as content diversity to avoid repetitiveness and give a sense of novelty throughout playing the game.

3.2 Requirements Analysis

The following is a requirements analysis following the AtLarge design framework (19). The user referred to in our requirements is the user of the PCG system, i.e. the developer of the game, and not the player of the game that is using the PCG system.

We formulate these requirements using a knowledge-based approach, beginning with an initial brainstorming phase and progressing through multiple cycles of refinement and adaptation. We form these requirements based on our existing knowledge and expertise in PCG systems and video game development, partially gained by reviewing scientific literature on PCG, and the construction of a reference architecture for PCG systems in games. Then, throughout the design and implementation of our prototype (see Section 4), we iteratively evolve our requirements. This iterative approach allows for the requirements to evolve organically.

3.2.1 Functional Requirements

The following is the list of functional requirements the system should satisfy. The requirements are formulated as user stories.

FR-1 As a user, I want to generate content, so that I can reduce manual content creation work, speeding up development, in addition to increasing replayability.

FR-2 As a user, I want to set constraints on the generated content, so that I can control the characteristics of the generated content.

FR-3 As a user, I want separate requests for content to generate different content, so that I can generate diverse content.

FR-4 As a user, I want to provide a seed value to the system that lets me receive the same content for the same seed, to enable reproducibility.

FR-5 The system must support content generation both locally and in a serverless environment, providing developers with flexible deployment options. This choice impacts performance and cost depending on the target platform and the complexity or volume of the generated content. It is important that the system supports both of these modes of operation, as it gives users of the system flexibility based on the needs of their application, which promotes adoption. Furthermore, it aids in the evaluation of the performance impact of serverless content generation.

FR-6 The system must support interchangeable content retrieval policies, allowing developers to adapt retrieval strategies based on varying performance and content diversity requirements. This NFR further improves flexibility of the system, such that it can more easily adapt to application requirements.

3.2.2 Non-functional Requirements

The following is a list of non-functional requirements that the system's design should adhere to.

NFR-1 The overhead of a serverless response shall not be larger than 2000 ms. Acceptable loading times differ based on the type of content generated and when the content should generate in the game (after a loading screen, during gameplay). For acceptable Quality of Experience, level load times should be kept under 33 seconds (33). Since content generation is only a fraction of level load time, and the serverless part of our system only generates the abstract representation of the generated content, the overhead threshold of the serverless response should be a fraction of the 33 seconds threshold. A threshold of 2000 ms, or approximately 6% of the 33-second upper bound, provides a reasonable and conservative performance requirement that leaves room for other stages of both the content pipeline (e.g., concrete instantiation, rendering) and any remaining application-specific loading steps.

NFR-2 The system shall be capable of handling an average of 200 content generation requests per second (RPS) during peak load without performance degradation. This threshold is based on a workload analysis of *Helldivers 2*, a recent game that uses procedural map generation. Its estimated peak player activity translates to

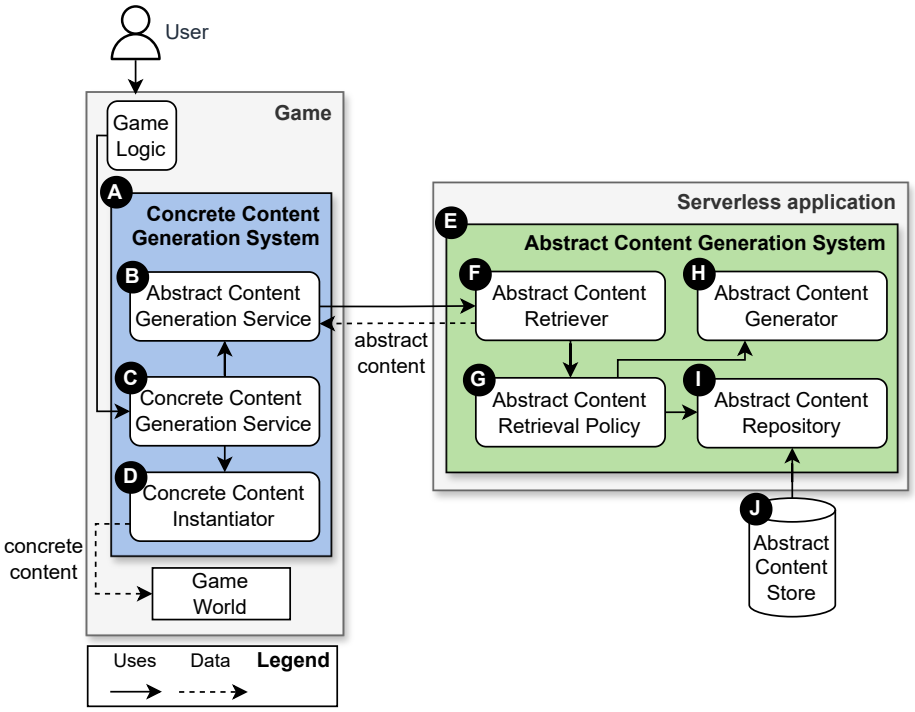


Figure 3.1: Overview of the design of ProcGen

.

approximately 200 RPS when accounting for average session lengths and cooperative play. This rationale and the calculation behind this threshold is further explained in Section 5.4.

NFR-3 The system shall be designed using a modular architecture to increase the maintainability and extensibility of the system, easing the introduction of new content generation methods and content types. This requirement is hard to quantify directly as it is a structural property that is better assessed through qualitative inspection of the design.

3.3 Design Overview

Figure 3.1 depicts the high level design of ProcGen. At the core of the design of ProcGen lies a fundamental architectural decision: the separation of the system into two subsystems, the *Concrete Content Generation System* (CCGS) and the *Abstract Content Generation System* (ACGS). In this design, the first subsystem (CCGS) takes the abstract content

representation generated by the latter (ACGS) to instantiate the concrete representation of the content. This design decision has several benefits. It minimizes network payload sizes and reduces transmission cost, as only the essential structural information of the content is sent, rather than fully generated assets. Furthermore, it makes for more efficient content storage, as only abstract content representation need to be stored in the data store. Additionally, abstract and concrete subsystem implementations can change independently, as long as they maintain the same interface, increasing extensibility. This allows developers to introduce new content generation techniques at the abstract level, without altering the concrete instantiation logic, or vice versa.

The CCGS (Ⓐ) is responsible for transforming the abstract content representation into its concrete form. This subsystem runs on the machine of the end-user playing the game, as it is responsible for instantiation of the generated content into the game world of the player. Some examples of concrete representations of generated content are terrain, vegetation, NPCs, or weapons. For more examples, see Section 2.1.2. The CCGS is invoked by the game logic, such as when the player enters a certain area or an in-game timer has elapsed.

The game logic interfaces with the CCGS through the *Concrete Content Generation Service* (Ⓒ), which it can pass some configuration or parameters to constrain or customize the generation process. This allows the user of the CCGS to, for example, generate more difficult content based on the player's progression, by passing in a different configuration that corresponds to content of a higher difficulty. The *Concrete Content Generation Service* is responsible for orchestrating the content generation process on the CCGS' side, by making calls to both the *Abstract Content Generation Service* (Ⓑ) with the given configuration to generate the abstract content, as well as the Concrete Content Instantiator (Ⓓ), by passing it the generated abstract content.

The *Abstract Content Generation Service*(Ⓑ) is responsible for setting up the request to the serverlessly deployed *Abstract Content Generation System* (ACGS) (Ⓔ), depicted in green. It makes sure that the request is in a format that can be accepted by the ACGS, for example a HTTP POST request with a properly formatted and validated JSON request body. Once the ACGS has received the content generation request and has completed the generation process (explained in the next paragraph), the generated abstract representation of the content is returned to the *Concrete Content Instantiator* (Ⓓ). This component is responsible for the actual transformation from abstract to concrete representation. This concrete content is then instantiated, or spawned, into the *Game World*.

As mentioned previously, for the CCGS (Ⓐ) to obtain abstract content that it can transform into its concrete representation, it makes calls to the Abstract Content Generation

System (ACGS) (Ⓔ). The ACGS runs as a serverless application (although it can run on the same machine as the CCGS, and even in the same game engine). Requests can be made to the ACGS for abstract content, with the option to pass parameters with the request for constraints on the generation process. Incoming requests are received by the ACGS and handled by the *Abstract Content Retriever* (Ⓕ). This component is responsible for orchestrating the retrieval process of the abstract content. Retrieval in this context means using a retrieval policy (Ⓖ) to either generate new content or fetch previously generated abstract content from a data store.

These policies differ in their logic on when to generate or fetch, and each policy has different use cases and performance implications (see Section 3.4 for a description of the retrieval policies designed in this thesis). In the case when new content needs to be generated, the retrieval policy calls the *Abstract Content Generator* (Ⓗ) to generate new content. The generator employs a content generation method to generate the content (see Section 2.1.3 for more background on different content generators). This content generator should generate content in an abstract content representation. Then, based on the logic of the policy used, the generated content can be stored in a data store, such that it can be re-used in subsequent requests. Alternatively, when the retrieval policy decides to fetch previously generated content, it uses the *Abstract Content Repository* (Ⓘ) to fetch the content from an *Abstract Content Store* (Ⓙ). The repository can do so by querying the data store based on parameters given through the request that was sent by the CCGS. Whether the content's requirements satisfy the constraints given by the parameters can be influenced by the strictness of the policy used. In the case that the data store does not contain satisfying content, the policy can choose to generate the content after all.

The type of data store used ((Ⓙ) in Figure 3.1) is not enforced by the design. Choosing a fitting data storage solution depends on several factors. One factor being the frequency of reads from the data store and another being the frequency of writes to the data store. Another factor is the query requirements in terms of complexity, i.e. the number of fields queried simultaneously. For content that has multiple dimensions in terms of content characteristics, the storage system should support querying on multiple dimensions. The impact of each of these factors is also influenced by the content retrieval policies used. An extreme example would be for an implementation that only uses retrieval policies with a fetch probability of 0%, meaning no data store would ever be used and content would always be generated on the fly, thus no data store would be required.

When the *Abstract Content Retriever* (Ⓕ) has used the policy to go through the process of retrieving content that satisfies the constraints, it returns the abstract content to the *Ab-*

Policy	Fetch Probability	FitnessThreshold
Always Generate, Low Match	0.0	0.5
Always Generate, Medium Match	0.0	0.8
Always Generate, Exact Match	0.0	1.0
Balanced Fetch, Low Match	0.5	0.5
Balanced Fetch, Medium Match	0.5	0.8
Balanced Fetch, Exact Match	0.5	1.0
Always Fetch, Low Match	1.0	0.5
Always Fetch, Medium Match	1.0	0.8
Always Fetch, Exact Match	1.0	1.0

Table 3.1: Retrieval policies, with their configurations in terms of *fetch probability* and *fitness threshold*.

stract Content Generation Service (Ⓑ), which returns it to its caller, the *Concrete Content Generation Service* (Ⓒ). As discussed previously, this service passes the abstract content to the *Concrete Content Instantiator* (Ⓓ), after which the entire process is complete and the generated content has spawned in the *Game World*.

3.4 Content Retrieval Policies

The policies designed in this work are outlined in table 3.1 , and differ in terms of two properties: *Fetch probability* and *Fitness threshold*.

Fetch probability is the chance that the policy will attempt to fetch existing content from the data store, rather than generate new content. A fetch probability of 0 indicates that content will always be generated, while a value of 1 implies that the content will always be fetched if it meets the relevant criteria. Intermediate values represent a balanced approach where fetching occurs probabilistically. Policies with a fetch probability > 0 are considered 'pre-generation-based policies', as they make use of pre-generated content, with the goal to reduce the latency of the system.

Fitness threshold is the minimum fitness score for a content item to be eligible to be returned to the user. Fitness score is a function $F(A, B)$, where A and B represent the characteristics or properties of a content item. The function produces a score between 0 and 1, with higher values indicating a closer match between the items. For example, when a fitness threshold is set at 0.8, a content item will only be fetched if the score between its properties and those specified in the request is 0.8 or greater. This is a domain-specific function, dependent on the content and its characteristics.

These two properties were chosen to be able to influence several performance metrics of the PCG system, namely latency, fitness score, and duplicate rate. The importance, or acceptable range, of these metrics varies between applications. The latency indicates the time between sending the request to the system and receiving a response from the system. This metric is important as content generation can be time sensitive. The fitness score indicates how close the content returned by the system is to the parameters in the request to the system. Some applications have strict criteria for content characteristics, e.g. when a game's difficulty is influenced by the retrieved content, which means the fitness score of retrieved content is of importance. The duplicate rate is another performance metric, which indicates the number of duplicate content items in a set of requests. This can be important for applications that require novelty in their content, for example to increase replayability and novelty of the games. Experiment 5.5 explores the impact of the different policies outlined in Table 3.1 on these metrics.

For each policy, each time content is generated, the content may be persisted to the data store. In a production environment, this behavior might be desirable to allow the data store to grow over time, progressively enhancing the range of cached content. However, in an experimental setup, this behavior can be less desirable to ensure that the data store stays in a fixed state over the duration of the experiment. It is important to note that if this behavior is implemented, the data store would grow indefinitely. As such, a pruning mechanism would be required to manage the data store's size. However, such mechanisms are outside the scope of this design.

3.5 Considerations for Serverless PCG Design

This section examines how serverless computing and procedural content generation interact to address key goals in designing a scalable and efficient serverless PCG system. These goals include reducing latency, enhancing scalability, and supporting diverse content generation methods and types. By analyzing these interactions, we derive actionable insights to inform the high-level system design.

Reducing Latency

Reducing latency of PCG systems in games is important due to the real-time nature of games. Serverless computing introduces both challenges and opportunities in minimizing latency.

Serverless functions often experience cold starts when no functions instances are available. This initialization delay can be problematic for time-sensitive PCG tasks. To partially mitigate this, pre-generation and caching of content can be used to reduce latency.

To minimize data transfer overheads, abstract content representations can be used which are smaller in size, as opposed to transferring concrete content representations over the network, which can be significantly larger in size due to assets needing to be transferred. By offloading the abstract generation task to the serverless back-end while minimizing the size of transmitted data, the system balances responsiveness with efficiency.

Enhancing Scalability

Games and PCG systems often face unpredictable workloads, such as spikes during game launch events or user driven content generation surges. Serverless computing's elasticity aligns well with these scalability needs.

Serverless platforms dynamically allocate resources on demand, ensuring that bursty workloads do not overwhelm the system. In addition, a steady increase in request volume over longer periods of time result in automatic scaling, requiring minimal to no manual maintenance.

Serverless computing employs fine-grained billing, charging only for actual usage. This allows the PCG system to scale cost-effectively, generating content during periods of high demand without maintaining idle resources during low-activity periods.

Supporting Diverse Content Generation Methods and Types

The design of a serverless PCG system must accommodate a wide range of content generation methods and types, each with unique computational and resource requirements.

Constructive methods generate content deterministically in fixed time, making them ideal for serverless deployment. Their predictable resource usage ensures compatibility with runtime and memory constraints of serverless platforms.

In contrast, methods that require iterative computation such as search-based methods pose challenges for serverless deployment due to runtime limits. Ensuring a limit on the runtime of these methods and employing pre-generation and caching mechanisms can overcome these constraints.

Using abstract representations of content (e.g., symbolic dungeon layouts) allows for efficient transmission and flexible client-side instantiation. This separation of abstract and

concrete content generation ensures that diverse content types can be supported without overwhelming serverless or client-side resources.

Implications for Design

The above considerations influence the high-level design of the system. To minimize latency, pre-generation or caching of generated content can be used, in combination with policies that encapsulate the logic of fetching versus generating and storing content. Furthermore, by only generating abstract content representations in the serverless environment and transforming the abstract representation into concrete assets on the client-side, we ensure efficient data transmission and a balance between offloading on client-side computation. Lastly, a modular architecture ensures extensibility to diverse PCG methods and content types.

3.6 Design Alternatives

This section describes the design alternatives and the reasoning for choosing our approach, opposed to these alternatives.

Deployment Models

A fundamental decision made in the design of this system is offloading PCG content generation to a serverlessly deployed system. While the choice of serverless is central to our first research question (RQ-1 How to design a serverless procedural content generation system?), it is worth exploring alternative deployment models and their trade-offs.

One option is to use dedicated virtual machines in a cloud or on-premise environment. The benefit of this is full control over the OS, runtime environment, and software stack, allowing for fine-tuned optimizations. Additionally, costs are more predictable since they are constrained to the VM's configuration. Moreover, a VM-based system is always active, eliminating the latency spikes caused by cold starts in serverless environments. This approach also has drawbacks, namely, scalability is limited, as scaling requires manual provisioning. Furthermore, maintenance is higher as the user is responsible for the management of the operations, which means regular patching, monitoring and configuring the server. Another drawback is resource inefficiency, since resources are allocated regardless of whether they are fully utilized, potentially increasing costs during idle periods.

A deployment model with scalability characteristics closer to a serverlessly deployed system is container orchestration. This approach makes the ACGS containerized and deployed

in a cluster of containers, leveraging features such as automated scaling and resource management. This approach has the benefit of horizontal scaling, similar to serverless, where containers are spun up or down depending on demand. This also makes the approach more cost effective, as the available infrastructure can more closely match demand, meaning less cost and wasted resources. Furthermore, container orchestration tools often provide built-in features for fault tolerance, offering high availability. A disadvantage of this approach is the operational complexity that it requires, where the container orchestration tool needs to be set up, managed and monitored, as well as the maintenance of the container images themselves. Similar to serverless, containers require start up time if they are scaled down during idle periods. Additionally, while cost effective, container orchestration can incur a higher baseline cost compared to serverless functions, especially for small workloads.

Ultimately, the serverless deployment model was chosen not only because of our research focus but also due to its inherent advantages. Serverless architectures excel at dynamically scaling to meet fluctuating demand, significantly reducing operational overhead. Additionally, their pay-as-you-go pricing model ensures cost efficiency, particularly for workloads with bursty or unpredictable usage patterns.

Separation of Abstract and Concrete Content Generation

Our design splits abstract and concrete content generation as separate subsystems, where the abstract content generation system runs serverlessly and the concrete content generation system runs on client-side. An alternative approach would be to combine both tasks into the same system. This would mean that the both abstract and concrete generation happens on the serverlessly deployed system, and the concrete content would be sent over the network to the client. The benefit of this approach is that all generation-related computation would be offloaded to the serverlessly deployed system, reducing client-side computational overhead.

While this approach reduces the client-side workload in terms of computation required for generation, it introduces several trade-offs that affect system performance and flexibility. Firstly, due to the increase in offloading, the response time of the deployed system will increase. This is due to the PCG system needing to transform the abstract content into its concrete format, as well as due to the size increase of the payloads, which now include the entire generated/placed assets. Depending on the content type, concrete generation payloads could grow by an order of magnitude, from kilobytes to megabytes, due to the size and type of assets (e.g., textures or 3D models). Secondly, there are cost concerns, as the increased runtime due to waiting on the generation of fully generated assets will lead to a

higher cost. Lastly, centralizing concrete generation might reduce the flexibility developers have to tweak or adapt the content, as altering the content in its abstract format is easier than when the content is already in its concrete asset form.

Chapter 4

Implementing a Prototype of ProcGen

In this chapter, we present the implementation of our prototype of the design for a serverless procedural content generation system. The implementation follows the design that was outlined in Chapter 3. We implement our design as a Unity package that is able to generate two-dimensional dungeons.

4.1 Dungeon Generation Requirements

This section describes the requirements of our implementation of a prototype that follows the design that was outlined in Chapter 3. In addition to having the same requirements as the design, we have created requirements specific to the prototype.

FR-DG-1 The system should create dungeons from user-created rooms by procedurally connecting rooms.

FR-DG-2 The user should be able to specify a desired dungeon depth (number of rooms to reach the final room).

FR-DG-3 The user should be able to specify the desired dungeon branching depth (depth of the number of rooms branching off of the main path).

FR-DG-4 Dungeons should have an abstract and a concrete representation.

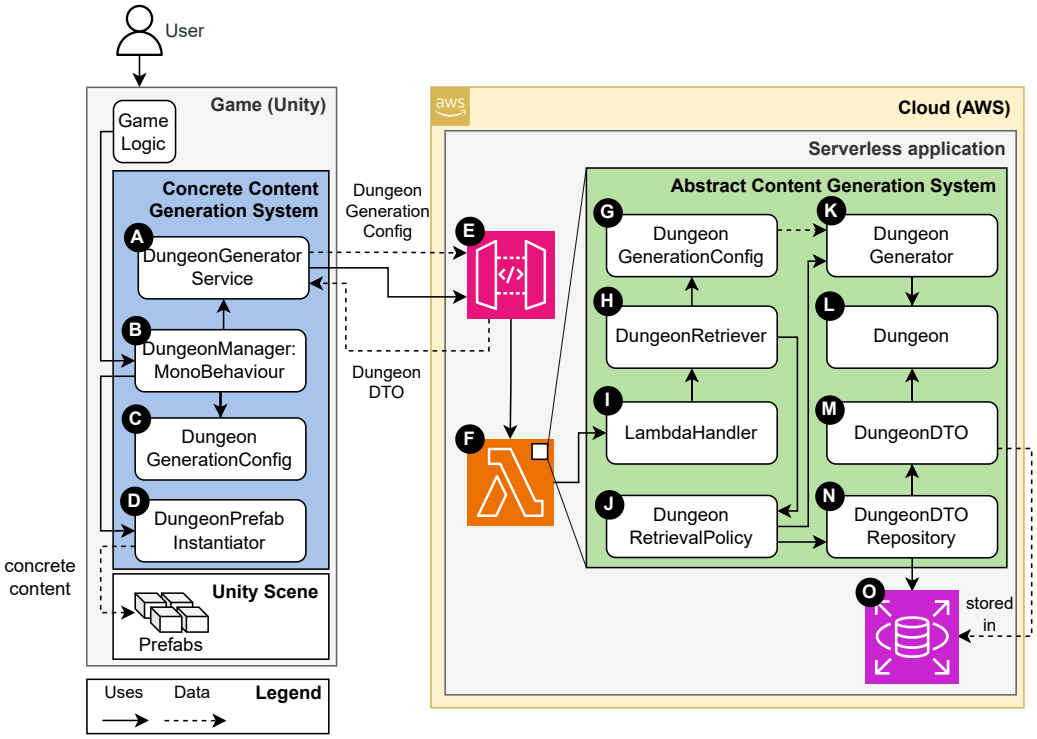


Figure 4.1: An overview of our prototype of ProcGen.

4.2 Overview of ProcGen

Our prototype, named ProcGen, is shown in Figure 4.1. ProcGen generates 2-dimensional dungeons, where the user can specify the desired number of rooms on the main path and the branching depth. The main path is defined as the number of rooms that need to be crossed in order to reach the end of the dungeon, starting from some room that is marked as the starting room. The branching depth is defined as the maximum number of subsequent rooms that are adjacent to any room on the main path. See 4.2.4 for more information on how ProcGen generates dungeons.

ProcGen is split up into two sub-systems, the Concrete Content Generation System (CCGS), depicted in blue, and the Abstract Content Generation System (ACGS), depicted in green. The CCGS exist in the Unity environment, which is running on the client, whereas the ACGS can run in both the Unity environment (i.e. locally), as well as serverlessly (i.e. remotely). When it is running serverlessly, the components, which are implemented as C# classes, still exist in the Unity package's code base.

4.2.1 Concrete Content Generation System

The CCGS is responsible for transforming the abstract representation of the content into its concrete representation. This concrete representation is in the form of prefabs in the Unity scene.

The *DungeonManager* (Ⓑ) is the only MonoBehaviour[1] in the system. MonoBehaviours are a fundamental component class in Unity that provide a way to add behaviour and functionality to GameObjects. In this case, the MonoBehaviour acts as a way for the user (developer) to interact with ProcGen, allowing configuration of the settings of the content generation system. The *DungeonManager* is responsible for orchestrating the generation of a dungeon by invoking the *DungeonGeneratorService* (Ⓐ) with a given configuration. The configuration is defined in the *DungeonGenerationConfig* (Ⓒ), which has a set of configuration options that are specific to the content generation algorithm and the content type it generates.

The *DungeonGeneratorService* (Ⓐ) is responsible for making calls to the *Abstract Content Generation System*. When ProcGen is running with a serverlessly deployed ACGS, these calls are done through HTTP POST requests, where the *DungeonGenerationConfig* is serialized into its JSON representation and put into the POST body of the request. In the case where the ACGS is running locally, the entry class of the ACGS is called directly. This entry class is the *DungeonRetriever* (Ⓗ), which is explained in more detail in subsection 4.2.2. The *DungeonRetriever* returns an abstract representation of a dungeon in the form of a *DungeonDTO* (Ⓜ), which is a serializable version of a Dungeon (Ⓛ).

When the DungeonGeneratorService (Ⓐ) has obtained a DungeonDTO, it returns it to the *DungeonManager*, which then calls the *DungeonPrefabInstantiator* (Ⓓ) to handle the instantiation of the prefabs by transforming the abstract representation of a Dungeon (Ⓛ) into a set prefabs instantiated in the Unity Scene.

4.2.2 Abstract Content Generation System

The ACGS (in green) is responsible for generating the abstract content representation, and can be run locally or serverlessly. On a high level, it takes as input the *DungeonGenerationConfig* (Ⓖ) (which is the same as (Ⓒ)) and returns a *Dungeon* (Ⓛ) in the form of a *DungeonDTO* (Ⓜ), which is a serializable version of a *Dungeon* that is able to be stored in a data store. This *Dungeon* is the data representation that is later to be transformed into Unity prefab instantiations (i.e. their concrete representation) by the CCGS.

[1] `https://docs.unity3d.com/Manual/class-MonoBehaviour.html`

In front of the ACGS is an *AWS API Gateway*[1] (E) which acts as an API front-end that routes incoming requests to our ACGS. The *API Gateway* is a serverless service which provides secure and scalable routing and event forwarding with built-in monitoring. The ACGS is deployed as a Lambda function (F), which triggers when the *API Gateway* receives a request on a specified endpoint. The *API Gateway* creates a corresponding event object and invokes the Lambda function. This happens through an internal AWS invocation mechanism. The Lambda function contains the code of our ACGS, where the incoming event is handled by the *LambdaHandler* (I). This handler transforms the event object into a *DungeonGenerationConfig* (G) and calls the *DungeonRetriever* (H) to generate or fetch a *DungeonDTO*. The *LambdaHandler* is only compiled when deployed in the serverless environment through conditional compilation pre-precessor directives in C#.

The *DungeonRetriever* (H) is responsible for handling requests for Dungeons and responding with a Dungeon that matches the constraints specified in the parameters for the request. These constraints are specified in the *DungeonGenerationConfig* (G) and the DungeonRetriever passes these constraints to the *DungeonRetrievalPolicy* (J) that it uses. The retrieval of DungeonDTOs in the retriever is done asynchronously as Dungeons can be fetched from a data store, depending on the retrieval policy used.

The *DungeonRetrievalPolicy* (J) uses a *DungeonGenerator* (K) and a *DungeonDTO-Repository* (N) to provide the *DungeonRetriever* with *DungeonDTOs* (M). This policy is specified in the *DungeonGenerationConfig*, and is concerned with balancing performance against content quality. The retrieval policy dictates the probability of generating a new dungeon using the generator, or fetching an existing dungeon using the repository (as the parameter *fetch probability*), as well as the strictness of how close the retrieved content should be to the constraint set in the DungeonGenerationConfig (as the parameter *fitness threshold*).

In the case where a new dungeon needs to be generated, a *DungeonGenerator* (K) is used. This component contains an algorithm to generate *Dungeons* and can easily be swapped out by a different Dungeon generating algorithm, as it uses the *IDungeon-Generator* interface (not depicted in Figure 4.1). The input to a *DungeonGenerator* is a *DungeonGenerationConfig* (G), which contains the input parameters used by the dungeon generation algorithm. In our case, this contains the desired depth of the main path, and the desired branching depth, as well as the room templates that should be used by the generator to generate the dungeon with. The output of the generator is a *Dungeon*

[1] https://aws.amazon.com/api-gateway/

(Ⓛ), which specifies where each dungeon room should be placed in terms of location and rotation.

Alternatively, a dungeon can fetched from the data store using the *DungeonDTORepository* (Ⓝ). This repository is responsible for encapsulating the logic of accessing the dungeons from a data store. This component provides functions to add *DungeonDTOs* to a data store, and fetch random *DungeonDTOs* from the data store, given a set of constraints such as desired main path or branching depth.

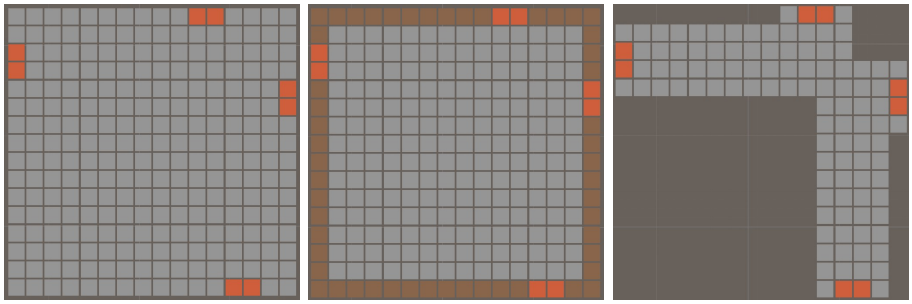
We store our *DungeonDTOs* (Ⓜ) in a relational database (but an implementation of a repository using a NoSQL database is also present in the prototype). This relational database is provided as a service on AWS named RDS (Relational Database Service)(Ⓞ). The *DungeonDTOs* contain the number of rooms on the main path, the desired branching depth, and the dungeon itself as a JSON string. Essentially, this data structure is as a set of properties describing the characteristics of the data, and the data itself. These characteristics are calculated once upon generation of the dungeon and stored with the dungeon record. To query the data, we query the characteristics, which is more efficient, as opposed to needing to calculate the characteristics at query-time.

4.2.3 Infrastructure as Code

This subsection explain how we employ Infrastructure as Code (IaC) in ProcGen. IaC is a way for developers to write the infrastructure required to deploy a system as code or configuration files. This configuration defines what the final state should be of the infrastructure for your application or use case, and the IaC tool will determine how to achieve that state. The benefit of this is that you do not need to manually set up infrastructure through, for example, graphical user interfaces in cloud environments, which is time consuming and error prone. Additionally, it allows you to version control your infrastructure configurations, where the IaC files can be stored in for example a Git repository. Furthermore, it is easier to replicate an infrastructure to improve consistency between deployments.

The IaC tool that we used is AWS Serverless Application Model (SAM)[1]. AWS SAM is an open-source framework that provides a way to define serverless resources such as AWS Lambda functions, API Gateway endpoints and databases in AWS. AWS SAM allows you to define all the infrastructure/resources needed to run an application in AWS in the form of *yaml* files. AWS SAM extends AWS CloudFormation[2], by providing a more simple

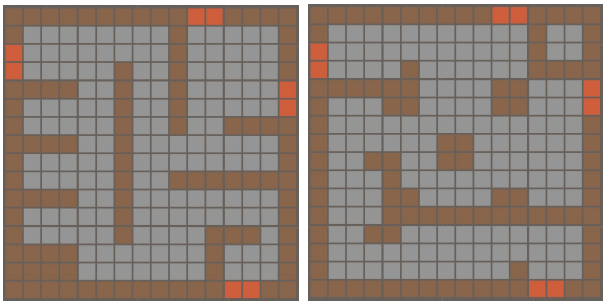
[1] https://aws.amazon.com/serverless/sam/
[2] https://docs.aws.amazon.com/cloudformation/



(a) The base room with floor tiles and doors, without any walls.

(b) The room with floor tiles, doors and outer walls only.

(c) A room with less floor tiles, but the same size as 4.2a and 4.2b.



(d) A variant of the above rooms, with an inner wall configuration.

(e) Another variant of the above two rooms, with an inner wall configuration.

Figure 4.2: Five dungeon rooms with the same size and door configuration. The gray cells are floor tiles, the orange cells doors, and the brown cells walls.

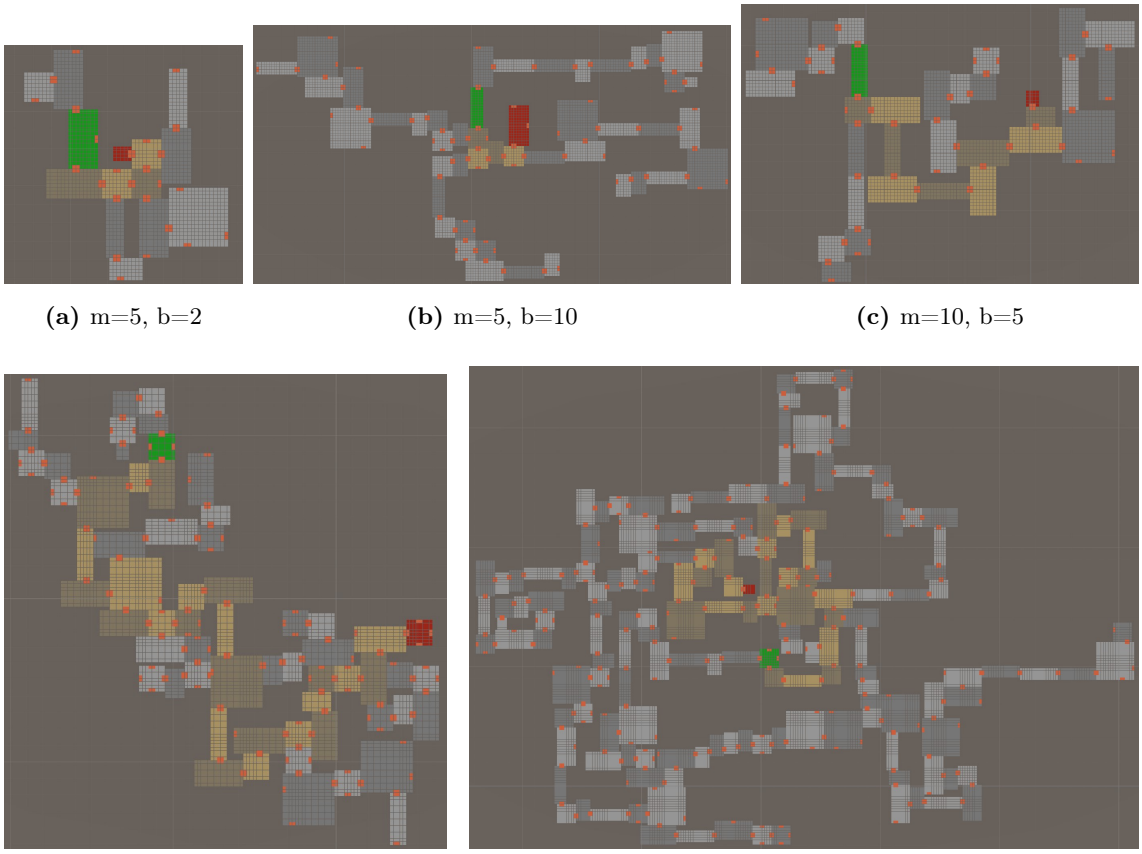
syntax with higher-level abstractions. AWS CloudFormation is a lower-level IaC service on AWS. We chose SAM over CloudFormation as it is simpler to use.

Our *yaml* template file for ProcGen consists of an *API Gateway*, a *Lambda function*, an *RDS Database*, and a set of policies required to perform the necessary actions to set up the infrastructure. More details of the function of these resources can be found in Subsection 4.2.2.

4.2.4 Dungeon Generation Algorithm

This subsection explains the dungeon generation algorithm that we use in ProcGen. This algorithm generates the abstract dungeon representation. An example of the abstract representation of a dungeon, alongside the corresponding concrete representation, is provided in Appendix 7.3 .

The algorithm that we implemented in our prototype is a constructive dungeon generation algorithm. Constructive algorithms construct their content step-by-step, as opposed to



(a) m=5, b=2 **(b)** m=5, b=10 **(c)** m=10, b=5

(d) m=25, b=5 **(e)** m=25, b=25

Figure 4.3: Five dungeons with various configurations, where **m** is the number of rooms on the main path (excluding the first room), and **b** is the desired number of rooms on the branch path. The entrance room is green, the exit room is red, the rooms on the main path are beige, and the rooms on branching paths are gray.

search-based algorithms, which generate content by exploring a solution space and evaluating candidates against a set of criteria. Constructive algorithms build content sequentially, ensuring that each addition is valid according to predefined rules, making them faster, more predictable and often times simpler.

Our dungeon generation algorithm is grid-based, meaning that each cell in a grid can have a value that represents some game element. In our implementation, we use two different elements: a door or a floor tile. A hard-constraint is that each room should have at least one door, where each door is made up of two adjacent door cells. Another hard-constraint is that each room should have its doors on the outer cells of the room. This makes it easier to connect rooms together, as rooms without this constraint risk overlapping their floor cells when aligning doors with adjacent rooms.

While we could have also included a wall element, we decided not to. The reason for this is that for a room with a certain configuration of floor tiles and doors, we can now create multiple variants of wall configurations. An example of this is given in Figure 4.2. The room in Figure 4.2a is an example of a room that we use for constructing dungeons. This could be seen to implicitly have walls around it, as seen in Figure 4.2b. This is not mandatory, however, and open outer-walls could have a different representation, e.g. an abyss, in specific game scenarios. Then, based on the open-space and door-based room of Figure 4.2a, we could 'fill-in' the walls in different configurations, as can be seen in Figure 4.2d and Figure 4.2e. These dungeon rooms with walls could be manually created as sub-templates, or generated through a separate algorithm step. However, in this work we have not implemented any system to either choose these sub-templates or generate them, as they are not required to have a valid dungeon and demonstrate our prototype.

The algorithm works as follows. Pre-designed dungeon rooms, again, consisting of door- and floor cells, are provided to the algorithm to use as templates. The user can provide the desired depth on the main path, the desired branching depth and the maximum number of retries as parameters to the algorithm. The desired depth on the main path represents how deep the dungeon should be in terms of rooms to traverse through from the entrance to the exit. For example, a dungeon with a desired depth of five requires the user or player to move through five doors in order to reach the exit room. This means the dungeon has a total of six rooms on the main path, an entrance, an exit, and four rooms in-between.

When the algorithm starts generating the dungeon, we first construct the main path. We do this by selecting a random room from the set of templates. Then, until the desired branching depth is reached, we keep selecting rooms from the set of templates and connecting them to the last placed room. Connection is done by lining up the 2-cell-wide doors that exist in each dungeon room template. To find a valid connection between rooms, we line up the selected room to the last placed room and check if their doors align. If not, we keep moving the selected room along the available door of the last placed room, rotating when all cells of a side have been tried, until a connection is found. If no connection is found, we randomly select another room from the set of templates. When the desired branching depth is reached, we mark the last room as the exit room.

Now that we have a 'main path', that is, a path from the entrance to the exit, we can work on constructing branching rooms. Branching rooms are rooms that form a path from doors that have no connection yet. These paths are thus connected to rooms on the main path that have three or more doors (where two are already connected in order to be part of the main path). For each available door, we use the same technique of selecting a random

room from the set of template rooms and trying to make a connection between the rooms. We keep repeating this until the desired branching depth is reached from the original available door from the main path. This procedure is then repeated for each available door on the main path. It can occur that at some point no room from the set of template rooms can be successfully connected, but the desired branching depth has not been reached yet. When this is the case, the algorithm disconnects the last placed room and continues with connecting randomly selected template rooms. We call this 'walking back', and when the algorithm consecutively fails to reach the desired branching depth, a counter increases. At certain thresholds, the number of steps to walk back increases, in an attempt to speed up the process when certain room connections are in place that make it very unlikely to ever reach the desired branching depth. The thresholds for the number of steps to walk back have been determined through trial and error.

4.2.5 Fitness Function

We implemented a fitness function, so that we can use it to filter dungeons retrieved from the database when the dungeon retrieval policy decides to fetch a dungeon, instead of generating it. Retrieval policies have a fitness threshold set that determines the minimum fitness score the dungeons in the data store should have in order to be deemed eligible for retrieval.

The fitness function (see Equation 4.1) takes as input the desired depth on the main and branching paths, and the dungeon's actual depth on the main and branching paths. The output of the fitness function is a fitness score between 0 and 1 that expresses how similar the dungeon is to the desired values. A fitness score of 1 has perfect fitness, i.e. a perfect match between the configuration and the dungeon.

$$\text{FitnessScore} = F(d_m, d_b, a_m, a_b) \tag{4.1}$$

where:

$$d_m = \text{desired main depth}$$
$$d_b = \text{desired branching depth}$$
$$a_m = \text{actual main depth}$$
$$a_b = \text{actual branching depth}$$

To calculate the fitness score (Equation 4.2), we first compute a normalized score between 0 and 1 for each dungeon configuration criterion: the depth score s_d and breadth score s_b. The average of these two scores is then taken to obtain the final fitness score.

$$\text{FitnessScore} = \max\left(0, \min\left(1, \frac{s_d + s_b}{n}\right)\right) \tag{4.2}$$

where:

s_d = depth score

s_b = breadth score

$n = 2$ (the number of criteria)

To calculate the score per criterion (see Equation 4.3), we take the deviation between the actual criterion (a) and desired criterion (d), and compare that against a max deviation value that is based off of the desired criterion's value and a maximum deviation factor. For example, for s_d, if the desired depth on the main path of 12, and the dungeon to compare against has a main path depth of 9, the deviation is their difference, namely $|9 - 12| = 3$. The maximum allowed deviation is computed by multiplying the desired value d with a configurable deviation factor f. The configurable deviation factor allows us to tweak how aggressive our fitness function is, where increasing the value would reduce the aggressiveness of our fitness function as the max increases. In our current implementation, we simply took a value of 1 for the max deviation factor. Consequently, our max deviation is equal to $12 * 1 = 12$. We normalize the deviation by dividing it by the maximum deviation (e.g., $9/12 = 0.75$) and subtract the result from 1, yielding a final score of $s_d = 1 - (9/12) = 0.25$

$$criterionScore = s = \begin{cases} 1 - \dfrac{|a - d|}{d \cdot f} & \text{if } |a - d| < d \cdot f \\ 0 & \text{otherwise} \end{cases} \tag{4.3}$$

where:

a = actual criterion value (e.g., number of rooms)

d = desired criterion value

f = maximum deviation factor (i.e., MAX_DEVIATION_FACTOR)

This fitness function is in theory able to support an arbitrary number of criteria by adding additional criterion scores to the equation, although our current implementation uses two (main path- and branching path depth). Additionally, the fitness function's aggressiveness is able to be tweaked via the max deviation factor, allowing for flexibility based on the requirements of the domain. An improvement on this fitness function could be a weight value per criterion, such that each criterion's weight on the final fitness score can be configured.

For a code implementation in C#, see Listing 4.1. This implementation directly mirrors the mathematical formulation in Equations 4.1, 4.2 and 4.3. In the code, the parameter *IDungeonMetadata dungeonDTO* contains the dungeon's scores (a_m, a_b). The *desiredDepth* (d_m) and *desiredBreadth* (d_b) are member variables of the class.

Listing 4.1: Function to compute fitness score of a dungeon in relation to a desired dungeon configuration.

```
public float GetFitnessScore(IDungeonMetadata dungeonDTO)
{
    float depthScore = CalculateScore(
        dungeonDTO.NumberOfRoomsOnMainPath,
        desiredDepth
    );
    float breadthScore = CalculateScore(
        dungeonDTO.DesiredBranchingDepth,
        desiredBreadth
    );
    return Math.Max(
        0,
        Math.Min(1, (depthScore + breadthScore) / 2)
    );
}

private float CalculateScore(int actual, int desired)
{
    float deviation = Math.Abs(actual - desired);
    float maxDeviation = (desired * MAX_DEVIATION_FACTOR);
    if (deviation >= maxDeviation)
        return 0;
    return 1 - (deviation / maxDeviation);
}
```

4.3 Constraints and implementation design considerations

We chose to implement the abstract content generation system in plain C#, thus not running in a Unity environment, for several reasons. Firstly, this approach ensures that the data format we send over the network is Unity/engine agnostic, in our case in JSON format. This means our ACGS could be utilized by other game engines, which would only need to implement the concrete content generation system. Secondly, by avoiding the initialization of a headless Unity instance, we reduce the cold start time of our Lambda function. This separation also offers development flexibility, as developers can focus on the abstract generation logic without needing Unity-specific knowledge. Moreover, it simplifies testing, enabling the use of standard C# testing frameworks to accelerate development cycles and improve reliability. Finally, this approach aligns with microservice architecture principles, promoting system modularity and maintainability.

In terms of data storage, initially, no specific type of data store was chosen as the *Abstract Content Store* (Ⓙ) in Figure 3.1. Earlier versions of the prototype used a NoSQL data store, as this is a common implementation for storing data for serverless applications (34). During development, our query requirements extended, as we needed to query on multiple content characteristics. Due to this change in requirements, the use of the NoSQL data store posed challenges, as querying on multiple fields has limited efficiency compared to SQL. We therefore changed our data storage technology to a relational database (MySQL). This allows us to easily query on multiple fields simultaneously. The drawback here is that our database no longer scales horizontally, and has lower availability than a NoSQL solution.

Chapter 5

Evaluating ProcGen through Real-World Experiments

This chapter answers our third research question: **How to evaluate and validate a serverless procedural content generation system?** Core to the evaluation of our system is testing whether we adhere to the requirements set in Section 3, especially the non-functional requirements. Additionally, our goal is to gain insights into when and how a PCG system can benefit from serverless.

5.1 Main Findings

This section describes the main findings of our evaluation.

MF-1 The system scales well under increased load. During load testing, response times initially spiked due to cold starts (2500 ms average, 4500 ms 95th percentile), but stabilized around 300 ms on average ($\tilde{5}$00 ms 95th percentile), even at high request rates (360 requests per second) from a workload modeled to reflect real-world conditions. (Section 5.4)

MF-2 Serverless PCG can reduce perceived generation time by up to 10x, when using pre-generation-based policies. We find that policies with higher fetch probabilities reduce perceived generation time due to latency hiding, with an average of 1000 ms latency for policies that do not fetch, versus an average of 100 ms latency for policies that always fetch. This benefit comes with trade-offs to both fitness score and duplicate rate. (Section 5.5)

MF-3 Retrieval policies with higher fetch probabilities result in lower latency, where a policy with 100% fetch probability achieves latencies in the hundreds of milliseconds. In contrast, purely generated content policies experience significantly higher latencies, ranging from several seconds to tens of seconds. (Section 5.5)

MF-4 We observe a strong inverse correlation between fetch probability and the achieved fitness score of the acquired content, for policies with a lower fitness threshold. Retrieval policies with higher fetch probabilities and lower fitness thresholds tend to reduce the fitness score of retrieved content. Policies with a 0% fetch probability consistently achieve a perfect fitness score of 1.0, as all content is generated from scratch. In contrast, non-zero fetch probabilities cause the fitness score to approach the policy's fitness threshold, with higher fetch probabilities bringing the score closer to the threshold. This is an expected result, because content generated for the specific request will meet the requirements of the request, whereas fetched content is selected from a cache with pre-generated generic content. (Section 5.5)

MF-5 Retrieval policies with a higher fetch probability increase the risk of returning duplicate results, increasing the duplicate rate. The impact of a pre-generation-based policy on duplicate rate can be partially diminished by increasing the size of the data store. (Section 5.5)

MF-6 Cold starts introduce increased latency and greater variability in response times (100-2600 ms), especially for workloads with requests in short bursts (0.25-0.5 seconds between requests), or workloads with spaced out requests (64-128 seconds between requests), as cold starts are more prevalent in these workloads. (Section 5.6).

MF-7 Serverless PCG that use pre-generation-based policies becomes more beneficial for performance when content generation time exceeds the overhead introduced by cold starts. This insight highlights the importance of considering the execution time of content generation when deciding between local and serverless environments (Section 5.6).

5.2 Overview of experiments to evaluate the system

This section gives an overview of the experiments that are executed to evaluate our system, listed in Table 5.1. These experiments aim to evaluate the non-functional requirements

Section	Objective	Mode of Operation	Workload Type	Testing Environment	Metric
5.4	Scalability via synthetic large-scale PCG workload	Serverless	Increasing users	CN, SG	Response time, throughput
5.5	Performance between retrieval policies	Serverless	Varying dungeons	SN, SG	Response time, fitness score, duplicate rate
5.6	Measure serverless overhead	Local, Serverless	Varying interval between requests	SN, SG	Response time

Table 5.1: An overview of the experiments used to evaluate ProcGen.
SN = Single-node headless Unity client, CN = Cluster of HTTP client nodes. SG = Serverless generator

NFR [1-5], which are defined in section 3.2.2. We use two experimental setups, described in more detail in section 5.3.

Our experiments differ in the mode of operation in which they run. Experiments 1 (Section 5.5) and 2 (Section 5.6) run in serverless mode only. Experiment 3 (Section 5.4) runs in both local and serverless mode, as its objective is to compare the performance of the two modes. Each experiment has a different workload, where experiment 1's workload increases the number of users (and thus requests) over time, experiment 2's workload varies the dungeon request configurations (desired depth and breadth of dungeons), and experiment 3's workload makes requests at different intervals. We have two different testing environments, where the testing environment for experiment 1 uses a cluster of nodes that make HTTP requests to the serverlessly deployed system, experiments 2 and 3 use a headless Unity client on a single node that makes HTTP requests to the serverlessly deployed system.

For each experiment, we measure a variety of metrics. Firstly, we measure the response time of the system in milliseconds (ms). Secondly, we indirectly measure the throughput by noting the requests per second (RPS). For experiment 2, we measure the fitness score of the requested dungeons and the duplicate rate.

5.3 Experimental Setup

This section describes the experimental setup for our experiments. Experiment 1 uses a cluster of VMs making HTTP calls to the serverlessly deployed abstract content generator

of ProcGen, without running Unity. Experiments 2 and 3 use a single headless Unity node as the client and ProcGen deployed serverlessly.

5.3.1 Experiment 1 Setup

For the serverless load testing experiments, our experimental setup is as follows. We use a set of VMs that run Locust[1], a (distributed) workload generation tool written in Python. These VMs will send HTTP requests to the deployed serverless PCG system. We can control the pattern of these requests through Locust. The HTTP requests are executed directly by Locust, and not via Unity. We have chosen to directly test the deployed serverless system through Locust, as opposed to testing it through Unity, as it allows us to (1) leverage existing load testing tools built outside Unity, and (2) isolate only the serverless part of the system, which is in this set of experiments the *system under test*. This allows us to gain more focused and accurate results, without the overhead of running Unity on the client side.

To make our experimental setup reproducible, we have automated infrastructure deployment through Terraform[2] and automated infrastructure configuration through Ansible[3]. This reproducible experimental setup is provided in a GitHub repository[4].

We run our experiments on a single-core node. Thus, we run one Locust worker instance per node. However, Locust can run thousands to tens of thousands of users per worker[5]. We run the experiments using three worker nodes.

As mentioned previously, our experiments run on AWS. AWS by default has a quota of 1000 concurrent execution environments. However, for new accounts this quota is set to a lower value of 10, which increases based on usage. Since we made a new AWS account for this thesis, we had to request a quota increase such that our max concurrent execution environments is equal to the default of 1000.

5.3.2 Experiment 2 and 3 Setup

For experiment 2 and 3, our experimental setup is as follows. We use a single headless Unity node. This node is used for running the experiments in both local and serverless mode. When the node is running locally, the node both requests content and generates it. When the node is running serverlessly, the node only requests content and the serverlessly

[1] `https://locust.io/`
[2] `https://www.terraform.io/`
[3] `https://www.ansible.com/`
[4] `https://github.com/misharigot/procgen-evaluation`
[5] `https://docs.locust.io/en/stable/running-distributed.html`

deployed system is generating/fetching content and sending it back to the node. The node is run on AWS' Elastic Compute Cloud (EC2), on a t2.micro. This node has 1 vCPU and 1GiB of RAM.

5.3.3 Reproducibility

Experiments in systems research are often comprised of complex interactions between components, where different configurations can lead to vastly different results. Reproducibility provides several benefits to a study. It allows other researches to more easily duplicate the study, which allows for validation of the results. This facilitates peer review and allows other researchers to more easily verify the claims of the study.

We have taken several measures to improve the reproducibility of this thesis. First, we have created an open source repository containing the code that evaluates our system. Second, the infrastructure used to evaluate our system is present as code, otherwise known as Infrastructure as Code (IaC). We implemented this using Terraform. Third, to automate the configuration of the deployed infrastructure, we use Ansible playbooks. These playbooks automatically install the required packages and run the experiments, after which they report the results. Lastly, the code that plots the graphs is also provided in the form of a Jupyter notebook written in Python. The aforementioned Terraform, Ansible and Python code are all present in the open source repository.

5.4 Experiment 1: Scalability via Synthetic Large-Scale PCG Workload

This experiment evaluates the scalability of our system when run serverlessly. Thus, we evaluate the abstract content generation system run in a serverless environment.

To calculate the concurrency required by our system, we can use the following equation:

$$Concurrency = R \times D \tag{5.1}$$

Where R is the average requests per second (RPS) and D is the average duration of each request. Since our workload has a peak load of 200 RPS, we can make an estimation on our required concurrency if we take, for example, an average request duration of 500 milliseconds. This would give us a required concurrency of $Concurrency = 200 \times 0.5 = 100$. In practice, during our experiment the max total concurrent executions was 95.

Number of users (peak concurrency)	Ramp up (users started/second)	Number of nodes	Wait time between requests in sec
2000	20	3	5 - 15 (10 avg)

Table 5.2: Overview of the experiment 1's configuration in Locust. We use a lower number of users than 450,000, but each user's average request interval is 10 seconds, instead of 20 minutes.

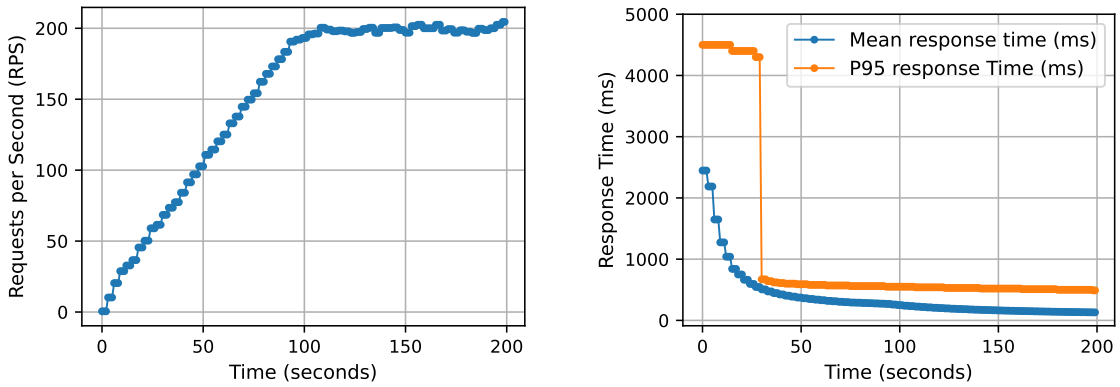
5.4.1 Experiment Design

Our experiment use a realistic workload that simulates a game that reaches a high number of concurrent users. This scenario can occur in the period after a game just launched and thus reaches peak popularity. We take the very popular recently released title Helldivers 2's peak concurrent users on Steam[1] as an example. The all-time peak concurrent users at the time of writing is around 450,000 users[2]. Helldivers 2 uses procedurally generated maps that are played in sessions ranging from 10 to 40 minutes, and on average around 20 minutes. These maps can be played individually or in a group of four players. We assume that each session generates a new map. Additionally, we assume that a fair portion of games will consist of teams, as the game has a cooperative nature, thus averaging team size at 2 and where the two players both share a single generated map. This results in 450,000/20/2 = 11,250 map generation requests per minute, or 11,250/60 = 187.5 requests per second. To evaluate this scenario, we use a workload of 200 requests per second.

We simulate a workload similar to the peak workload of Helldivers 2. To this end, we use the Locust configuration, described in Table 5.2 . While Helldivers 2 map requests have an average interval of 20 minutes between sessions, we want to shorten the duration of our experiment. Thus, we generate a comparable workload that ramps up to the peak of 200 requests per second by reducing the interval between requests to a range of 5 to 15 seconds, averaging 10 seconds. With a target of 200 requests per second, this corresponds to 2000 request over a 10-second interval, which we simulate using 2000 Locust users spread across three Locust worker nodes. Each Locust worker node increases its user count by 20 users per second. This gradual ramp-up allows us to observe patterns in the data over time. It would require a total of 100 seconds to reach 200 requests per seconds, or 2000 users.

[1] `https://store.steampowered.com/`

[2] `https://steamdb.info/app/553850/charts/`



(a) Requests per Second (RPS) over time.

(b) Mean and 95$^{\text{th}}$ percentile response time.

Figure 5.1: Experiment 1: Load testing our serverlessly deployed system.

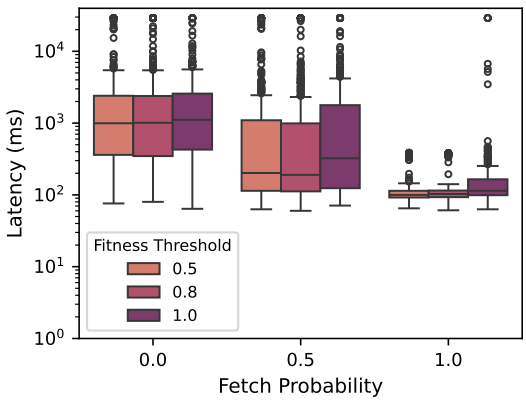
5.4.2 Results

Figure 5.1 presents the results of our scalability experiment. Figure (5.1a) shows the number of requests per second over time, while the Figure (5.1b) shows the response times in milliseconds during the same period.

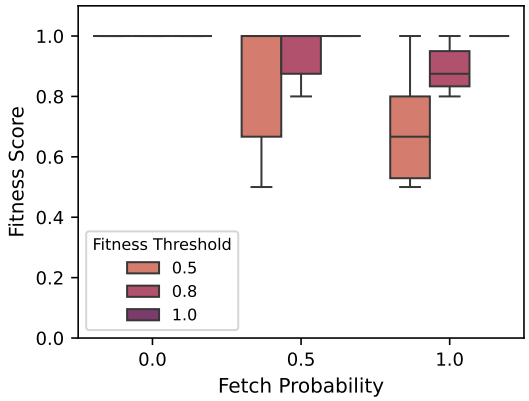
In Figure (5.1a), we observe that the system was able to handle the increasing request rate, ramping up to 200 RPS without any observed failures. The lack of errors highlights the system's reliability under load. The request rate stabilizes at around 200 RPS as expected, reflecting the target workload described in our experiment design.

The response times, shown in graph (5.1b), initially spike to around 4,500 milliseconds at the beginning of the experiment. This spike is caused by cold starts in the serverless environment, where the initial requests take longer due to the underlying infrastructure requiring time to scale up to meet demand. However, after this initial spike, the response times quickly stabilize at roughly 200 milliseconds on average, with the 95th percentile stabilizes around 500 milliseconds. This stabilization indicates that once the serverless infrastructure is warmed up, the system can handle high concurrent request loads efficiently. During this experiment, the total concurrent execution environments reached a total of 97.

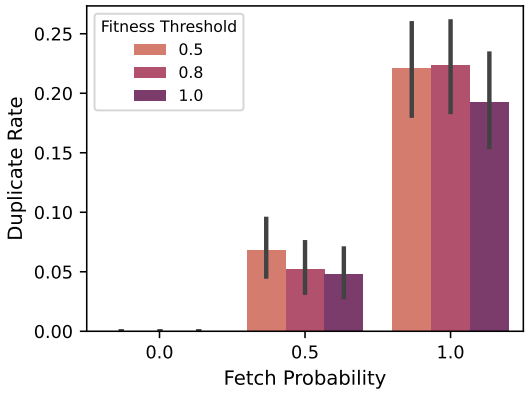
These results demonstrate that the system is able to scale well under high request rates, maintaining stable and predictable performance once the initial cold start period has passed. The system was able to handle 200 RPS without any failures, which is in line with the target workload modeled after a real-world game scenario like Helldivers 2. These findings indicate that the serverless architecture is suitable for supporting a procedural content generation system in games that expect a scaling workload with bursty or peak



(a) Latency measured using retrieval policies that differ in fetch probability and fitness threshold.



(b) Fitness score measured using retrieval policies that differ in fetch probability and fitness threshold.



(c) Duplicate rate using retrieval policies that differ in fetch probability and fitness threshold.

Figure 5.2: Experiment 2: Performance impact of retrieval policies on the metrics latency, duplicate rate, and fitness score

traffic patterns. However, developers should take into account the impact of cold starts in latency-sensitive games. This could be partially mitigated by configuring pre-warmed serverless execution environments.

5.5 Experiment 2: Performance Impact of Retrieval Policies

Our second experiment aims to assess the impact of different retrieval policies on the latency, fitness score, and duplicate rate of our PCG system.

5.5.1 Experiment Design

We use different content retrieval policies, which consist of two factors, their *fetch probability* and their *fitness threshold*. These policies are outlined in Section 3.4, in Table 3.1. The fetch probability is the chance that content is attempted to be retrieved from the data store, as opposed to being generated at the time of the request. The fitness threshold is the minimum fitness score that a content item should have to be eligible for selection by the retrieval policy. The fitness score is explained in subsection 3.4.

The workload consists of a uniformly distributed set of 10 requests per dungeon request configuration that ranges from a main path depth of [2, 10] (i.e, 8 different main path depths) and a branching depth of [1, 5] (i.e., 5 different branching path depths). The requests are made in intervals of 0.5 seconds, resulting in 8 * 5 = 40 unique dungeon request configurations requests. With 10 requests per configuration, we make a total of 400 requests.

The data store contains 10 unique dungeons per configuration. Thus, the data store has 400 unique dungeons, or 40 unique dungeons per dungeon configuration.

In terms of metrics, for each request, we measure the latency, fitness score, and duplicate rate. The latency indicates the time it takes between making a request for a dungeon to the system and getting a response from the system with dungeon. This is important for time-sensitive use cases. The fitness score of a dungeon indicates the degree to which the requested dungeon meets the criteria defined in the request. This is important when, for instance, the game's difficulty is influenced by the characteristics of the retrieved content. The duplicate rate counts the number of times a specific dungeon was already returned in a previous request in the set of requests in the experiment. This can be important when, for instance, duplicates would have an impact on the player's immersion, or the replayability of the game. We choose these three metrics to understand the trade-offs made when choosing a specific policy, where one policy could trade a high value in one metric for a lower value in another.

5.5.2 Results

Figures 5.2a, 5.2b, and 5.2c together illustrate how the choice of retrieval policy, defined by their **fetch probability** and **fitness threshold**, has an impact on the three key performance metrics: latency, fitness score, and duplicate rate.

Policies with **low fetch probabilities** consistently result in higher latencies, as generating content from scratch is computationally more intensive. This is especially clear for

policies with a 0.0 fetch probability, with average latencies around 1000 ms. However, these same policies excel at the metrics fitness score and duplicate rate. Generated dungeons always result in a fitness score of 1.0 due to the design of the generator, and their duplicate rate is 0.0, as they are always unique. The latency variability for both fetch probability of 0.0 and 0.5 is high, reflecting the varying complexity of requested dungeon configurations. The generation process is sensitive to the request parameters, where simpler dungeons (e.g., a short main path and shallow branching depth) take far less time to generate than complex ones.

In contrast, policies with **high fetch probabilities** achieve significantly lower latency. For instance, policies that always fetch have an average of 100 ms latency, in addition to a lower variance. This represents a tenfold improvement over purely generative policies. However, there is a trade-off for this low latency: the average fitness score decreases, as the retrieved content may not match as close to the requested content configuration, which holds especially true for lower fitness thresholds. In addition, duplicate rates increase under these policies, as retrieving the same content across requests becomes more likely.

The **fitness threshold** has a strong impact on the fitness score, especially for retrieval-heavy policies. Higher fitness thresholds restrict the set of eligible data in the store, leading to improved fitness scores, and somewhat reduced duplication. While the reduced duplication may seem counter-intuitive, it can be attributed to the fact that the stored dungeons are uniformly distributed across dungeon configurations/criteria. When applying a higher fitness threshold, retrieval is limited to a narrower portion of the data store, which reduces the likelihood of duplication. For policies that generate from scratch, the fitness threshold shows no influence, as our content generator uses a constructive generation approach and does not have the ability to generate loosely matching content. For higher fitness thresholds, a slight increase in latency can be observed, which we attribute to stricter filtering requirements.

These results highlight the inherent trade-offs between latency, fitness score, and content uniqueness in a PCG system that blends generation and retrieval using policies. Increasing fetch probability improves latency, at the cost of reduced fitness and increased duplication. Meanwhile, adjusting the fitness threshold allows for control over fitness score and duplication rate, but has limited impact on latency.

Users of ProcGen must therefore choose policies that align with their use cases. When low latency is a priority, policies with a high fetch probability are more fitting. For use cases where content must match the desired criteria more closely, or where uniqueness is

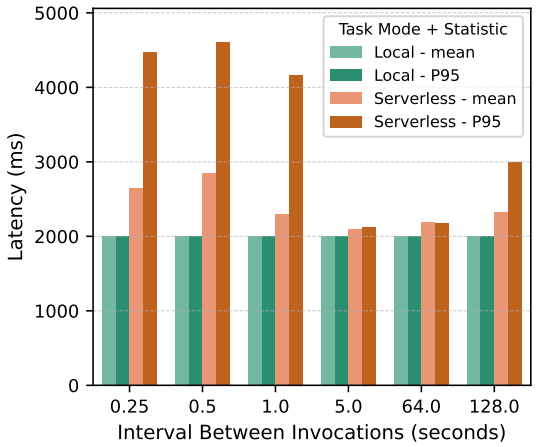


Figure 5.3: Experiment 3: Latency characteristics in local and serverless environments.

of importance, policies with higher generation rates and higher fitness thresholds are more appropriate.

5.6 Experiment 3: Performance Overhead of Serverless Content Generation

This experiment, illustrated in Figure 5.3, aims to understand both the overhead in content generation time for serverless execution, as well as the impact of cold start times.

5.6.1 Experiment Design

We request content from our system when running in both local and serverless mode, at varying intervals between requests. For each request, we measure the response time in milliseconds, otherwise known as the round-trip time (RTT). The x-axis plots the interval between the requests in seconds, and the y-axis shows the time it took for a dungeon to generate.

We execute 35 requests per benchmark, where each benchmark can run in either local and serverless mode, with a specified request interval. These request intervals range from .25 seconds to 128 seconds, with the intervals in between the minimum and maximum chosen such that we cover varying points along the range, without choosing intervals that would measure the same scenario. Our envisioned scenarios would be lower request intervals (.25 to 2 seconds) which could trigger cold starts due to concurrent executions, medium request intervals (5 seconds and 10 seconds) that could include warm starts for the majority of

the requests, and long intervals (60 seconds, 128 seconds), that could trigger cold starts due to execution environments being destroyed due to longer periods of inactivity between requests.

5.6.2 Results

For the local environment, we observe that the latency remains consistently close to 2000 ms across all intervals. The data for this group shows minimal spread, indicating that the response time is stable and predictable. These results align with our expectations, as we introduced a fixed delay of 2 seconds for each generation invocation.

For the serverless environment, we observe a performance overhead in latency ranging from approximately 100 ms (P95) without cold start times (e.g., the 5-second and 64-second intervals), to a maximum of 2600 ms (P95) for requests experiencing cold starts (e.g., the 0.5-second interval). Additionally, we observe significant variability in response times for the workloads with a smaller interval between requests (0.25s - 0.5s). This variability is significantly reduced for the 5-second and 64-second interval workloads, where performance becomes more stable. However, for the workload with a larger interval between request (128s), we observe moderate performance variability.

The observed pattern in performance variability supports our hypothesis that cold starts would occur in two scenarios: (1) when the interval between requests is very small, as concurrent executions will then take place in the scenario where a new requests is received while a previous request is still being processed, and no warm start can occur due to a lack of idle execution environments, and (2) when the interval between requests is large enough that execution environments are being destroyed due to inactivity, requiring re-initialization upon subsequent requests. These observations are in line with existing studies evaluating the cold start problem, specifically on AWS Lambda (35, 36, 37).

This performance variability highlights a key consideration when leveraging serverless computing: while it offers scalability and resource efficiency, it may not be ideal for latency-sensitive applications with inconsistent request patterns. These results suggest that optimization strategies, such as using provisioned concurrency, could help mitigate cold start issues. An additional insight from these results is that serverless PCG is particularly beneficial when local content generation time exceeds the latency overhead introduced by cold starts, in cases where the PCG system uses retrieval policies that fetch pre-generated content. In this scenario, the relative impact of cold start delays is reduced, as serverless execution remains faster than local generation.

Chapter 6

Related Work

This section discusses related studies that combine procedural content generation with serverless or cloud-based deployments. A brief description of each study is given, along with a comparison to our work.

The system described in (38) uses a distributed system for generating planetoids. The system consists of a central server processing gRPC or REST requests from clients for terrain segment generation, which are then sent through a communication channel implemented with Kafka as a message broker, which transmits the requests to a set of distributed worker servers. The worker servers are able to execute the requests in parallel to improve performance. Similar to our work, the task of generation is offloaded to the distributed system, leveraging cloud technologies to improve scalability and performance. In contrast, our work emphasizes ease of use with minimal operational overhead for developers, exploring trade-offs in latency, content quality and duplication, which is outside the scope of (38)'s work.

The authors of (39) present an offloading solution implemented in Unity to improve game performance. The system offloads game objects in Unity, where the decision to offload is determined heuristically based on the criteria *resource consumption*, *code dependency* and *network latency*. Offloaded game objects run on the server and their computed output (i.e., the resulting state of each game object) is sent back to the client, using a combination of Remote Procedure Calls (RPC) and the GigE Vision Streaming Protocol (GVSP). This approach bears some similarity to our work, where both systems aim to move computational load away from the client device to improve performance. Furthermore, their system is also implemented in Unity, separating concerns between local and remote components. However, their approach differs in that they offload entire game objects at runtime, whereas our work focuses on offload PCG tasks by dividing the system into an abstract and concrete

generation layer. Additionally, they offload using a heuristic approach at runtime, where our prototype implementation uses retrieval policies configured at compile-time. Finally, our system generates abstract representations remotely, which are instantiated at the client-side, whereas their system streams rendered output from server to client.

Kalva (40) presents a framework for adaptive game content generation systems in cloud gaming environments, leveraging distributed AI systems. The proposed framework consists of four main components: *AI processing modules* for content generation, a *content pipeline* responsible for asset management, *player analysis system* for behavior tracking, and a *delivery mechanism* for content distribution. The system uses AI methods for procedural content generation, such as convolutional neural networks for level generation, and natural language processing for narrative generation. While both Kalva's and our work propose high-level designs of PCG system running on cloud infrastructure, their focus lies on cloud gaming platforms such as Stadia and Xbox Cloud Gaming. In contrast, our system targets games running locally on the user's device, using serverless infrastructure solely for abstract content generation. Moreover, while their study provides a conceptual and architectural overview, we offer a detailed design, an open-source implementation, and an in-depth experimental evaluation of the system's performance.

Donkervliet et al. (18) propose a novel architecture for modifiable virtual environments (MVEs), such as the popular game Minecraft[1], that utilizes serverless to improve scalability and shift the developer's focus away from managing resources and scheduling, which in their vision could be the responsibility of cloud providers. In their vision, MVEs will become cloud-based services that scale to millions of concurrent players, where the MVE is split into multiple classes of services that are run as serverless functions, such as *virtual world services*, *game analytics*, *procedural content generation*, and *meta-gaming network services*. Our work fits in their vision, where the procedural content generation is modeled as a separate service, allowing it to scale independently based on demand.

In (17), a design for an MVE backend architecture is presented, that uses serverless computing to improve MVE scalability. Their system leverages serverless computing to offload procedural generation (terrain generation), among others. This allows terrain generation to scale horizontally, independent from the monolithic game server, similar to our work. The system caches and pre-fetches generated content based on the player's avatar, which indicates the likelihood of requiring certain terrain chunks in the near future due to the player's movement. While ProcGen also allows for fetching pre-generated content, our system does so based on retrieval policies chosen by the game developer. Thus, ours is a

[1]https://www.minecraft.net/en-us

more general approach, and game-specific retrieval policies similar to their system could be developed for ProcGen in the future, to provide the game developer with more control over performance metrics.

Chapter 7

Conclusion & Future Work

Procedural content generation in games offers a time- and cost-efficient, scalable solution to providing games with content, while enhancing replayability. Serverless computing, with its inherent scalability, fine-grained billing, and reduced operational overhead, presents an attractive alternative to traditional infrastructure. However, the intersection of serverless computing and game development, particularly in PCG, remains underexplored.

In this thesis, we designed, implemented a prototype of, and evaluated ProcGen, a serverless procedural content generation system. With this work, we identify opportunities and challenges of combining serverless with procedural content generation in games. By dividing the system up into two subsystems, where one focuses on abstract content generation, and the other on the transformation of this abstract format into content, we achieve a modular and scalable architecture. This chapter summarizes the contributions made in this thesis. We answer the research questions and outline future work.

7.1 Conclusion

RQ-1 How to design a serverless procedural content generation system?

Dividing a serverless procedural content generation system into two main subsystems — the Abstract Content Generation System (ACGS) and the Concrete Content Generation System (CCGS) — proved to be an effective architectural choice. The ACGS is responsible for the generation of content in its abstract representation, and the CCGS is responsible for transforming the abstract representation into game-specific assets. This separation of concerns allows for greater modularity and scalability. By offloading the computationally intensive tasks to a serverless

architecture, we can leverage the benefits of serverless, such as on demand scalability, high availability, minimal operational efforts and fine-grained billing. Our policy-based approach enables explicit control over trade-offs between latency, fitness score, and duplicate rate. This flexibility allows developers to adapt content retrieval behavior to different gameplay or performance priorities.

RQ-2 How to implement a serverless procedural content generation system?

We implemented our design for a serverless procedural content generation system as a prototype for dungeon generation as a Unity package, employing AWS serverless services, such as AWS Lambda, AWS API Gateway, and AWS SAM. Our prototype demonstrated that our design allows for an efficient and extendable system, able to generate content both in a serverless, as in a local environment. Through the use of AWS SAM, we reduced the operational efforts required to deploy the system.

RQ-3 How to evaluate and validate a serverless procedural content generation system?

To evaluate our designed and implemented system, we performed a set of experiments with a focus on measuring performance under various load conditions and configurations. The results showed that the system could handle high loads, successfully processing 200 requests per second, in line with our scalability requirements. Additionally, we observed that cold start times (ranging from 2200-3200 ms) are particularly relevant for shorter intervals (e.g., 0.25-0.5 seconds between requests) and longer intervals (e.g., 64-128 seconds) between requests, due to concurrency and the shut down of execution environments. Furthermore, retrieval policies showed significant trade-offs, as policies with higher fetch probabilities reached lower latencies of under 100 ms, but increased duplicate rates and reduced fitness score, compared to purely generative policies. One limitation of the evaluation was the synthetic nature of the load balancing experiments, which may not fully capture the complex nature of real-world usage patterns.

7.2 Future Work

To further validate the system, using it in a (large-scale) commercial game would be beneficial, as testing the system in real-world conditions would provide deeper insights. This could provide information on the performance, scalability and reliability of the system in a production environment.

This thesis focused on dungeon generation as a prototype for serverless PCG, but other content types could be explored. This would both validate the system and make the system more feature-rich, which would improve the chances of users adopting the system in their projects.

While we have implemented several content retrieval policies, there is room for further exploration on this end. For example, policies employing machine learning models based on usage patterns to dynamically adjust fetch probability and fitness thresholds could be explored, to improve the performance and content quality of the system. Additionally, policies that dynamically adjust based on system load and user demand could be investigated.

While this thesis' prototype is implemented as a Unity package, future work could explore integration with other game engines such as Unreal Engine or Godot. Due to the separation of a concrete- and an abstract content generation (sub)system, only the implementation of the concrete system in a different game engine is required, as the response of the abstract system is engine and language agnostic and thus the existing ACGS can be reused. However, implementing only the concrete subsystem in a different game engine would mean that the system can only run in serverless mode, as there is no abstract content generation system running on the end-user's machine.

Our system's content data store can currently be populated by policies that store content after they have been generated. Additionally, we perform experiments on our system when the data store is in a predetermined state in terms of data population. However, this state is reached artificially by manually inserting content items into the data store. In the future, our system could be extended to provide this data automatically through e.g. scheduled tasks. This would allow users of our system to not be dependent on usage patterns for their data store's population, which can have implications on the system's performance and the content quality/variety.

References

[1] **Get Newzoo's Newest Free Global Games Market Report 2023**, November 2023. 1

[2] Frankie Karrer. **Global Streaming Revenues Will Increase by 14% in 2023**, December 2023. 1

[3] Dylan Smith. **Recorded Music Industry Revenue Grew 10.2% in 2023: Report**, March 2024. 1

[4] Matthijs Jansen, Jesse Donkervliet, Animesh Trivedi, and Alexandru Iosup. **Can My WiFi Handle the Metaverse? A Performance Evaluation Of Meta's Flagship Virtual Reality Hardware**. In *Companion of the 2023 ACM/SPEC International Conference on Performance Engineering*, pages 297–303, Coimbra Portugal, April 2023. ACM. 2

[5] John David N. Dionisio, William G. Burns Iii, and Richard Gilbert. **3D Virtual Worlds and the Metaverse: Current Status and Future Possibilities**. *ACM Computing Surveys*, **45**(3):1–38, June 2013. 2

[6] Yuntao Wang, Zhou Su, Ning Zhang, Rui Xing, Dongxiao Liu, Tom H. Luan, and Xuemin Shen. **A Survey on Metaverse: Fundamentals, Security, and Privacy**. *IEEE Communications Surveys & Tutorials*, **25**(1):319–352, 2023. 2

[7] Mark Hendrikx, Sebastiaan Meijer, Joeri Van Der Velden, and Alexandru Iosup. **Procedural Content Generation for Games: A Survey**. *ACM Transactions on Multimedia Computing, Communications, and Applications*, **9**(1):1–22, February 2013. 2, 8, 9

[8] Breno M. F. Viana and Selan R. dos Santos. **A Survey of Procedural Dungeon Generation**. In *2019 18th Brazilian Symposium on Computer Games and*

Digital Entertainment (SBGames), pages 29–38, Rio de Janeiro, Brazil, October 2019. IEEE. 2, 9

[9] Noor Shaker, Julian Togelius, and Mark J. Nelson. *Procedural Content Generation in Games*. Computational Synthesis and Creative Systems. Springer International Publishing, Cham, 2016. 2

[10] Alexandru Iosup. **POGGI: Generating Puzzle Instances for Online Games on Grid Infrastructures**. *Concurrency and Computation: Practice and Experience*, **23**(2):158–171, 2011. 2

[11] Julian Togelius, Georgios N. Yannakakis, Kenneth O. Stanley, and Cameron Browne. **Search-Based Procedural Content Generation: A Taxonomy and Survey**. **3**(3):172–186. 2, 9

[12] Mojang Studios and Xbox Game Studios. **Minecraft Official Site**, 2011. Sandbox survival video game. 2

[13] Re-Logic. **Terraria**, 2011. 2D sandbox action-adventure game. 2

[14] Hello Games. **No Man's Sky**, 2016. Exploration and survival game. 2

[15] Iron Gate Studio. **Valheim**, 2021. Survival and exploration game inspired by Norse mythology. 2

[16] Arrowhead Game Studios. **Helldivers 2**, 2024. Third-person cooperative shooter. 2

[17] Jesse Donkervliet, Javier Ron, Junyan Li, Tiberiu Iancu, Cristina L. Abad, and Alexandru Iosup. **Servo: Increasing the Scalability of Modifiable Virtual Environments Using Serverless Computing – Extended Technical Report**, April 2023. 2, 14, 52

[18] Jesse Donkervliet, Animesh Trivedi, and Alexandru Iosup. **Towards Supporting Millions of Users in Modifiable Virtual Environments by Redesigning Minecraft-Like Games as Serverless Systems**. 2020. 2, 14, 52

[19] Alexandru Iosup, Laurens Versluis, Animesh Trivedi, Erwin van Eyk, Lucian Toader, Vincent van Beek, Giulia Frascaria, Ahmed Musaafir, and Sacheendra Talluri. **The AtLarge Vision on the Design of Distributed Systems and Ecosystems**, February 2019. 4, 15, 16

[20] **misharigot/procgen: Procgen, Procedural Content Generation in Unity.** https://github.com/misharigot/procgen. 5

[21] Kate Compton. **Practical Procedural Generation for Everyone**, May 2017. 7

[22] Misha Rigot, Jesse Donkervliet, and Alexandru Iosup. **A Reference Architecture for Procedural Content Generation Systems in Games**. 2023. 9

[23] Damian Kutzias and Sebastian von Mammen. **Recent Advances in Procedural Generation of Buildings: From Diversity to Integration**. *IEEE Transactions on Games*, pages 1–20, 2023. 9

[24] Barbara De Kegel and Mads Haahr. **Procedural Puzzle Generation: A Survey**. **12**(1):21–40. 9

[25] Jialin Liu, Sam Snodgrass, Ahmed Khalifa, Sebastian Risi, Georgios N. Yannakakis, and Julian Togelius. **Deep Learning for Procedural Content Generation**. **33**(1):19–37. 9

[26] Daniele Gravina, Ahmed Khalifa, Antonios Liapis, Julian Togelius, and Georgios N. Yannakakis. **Procedural Content Generation through Quality Diversity**. In *2019 IEEE Conference on Games (CoG)*, pages 1–8. 9

[27] Pittawat Taveekitworachai, Febri Abdullah, Mury F. Dewantoro, Ruck Thawonmas, Julian Togelius, and Jochen Renz. **ChatGPT4PCG Competition: Character-like Level Generation for Science Birds**. 9

[28] Noor Shaker, Miguel Nicolau, Georgios N. Yannakakis, Julian Togelius, and Michael O'Neill. **Evolving Levels for Super Mario Bros Using Grammatical Evolution**. In *2012 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 304–311, Granada, Spain, September 2012. IEEE. 9

[29] Vincent Breault, Sébastien Ouellet, and Jim Davies. **Let CONAN Tell You a Story: Procedural Quest Generation**. *Entertainment Computing*, **38**:100422, May 2021. 9

[30] Samuel Kounev, Nikolas Herbst, Cristina L. Abad, Alexandru Iosup, Ian Foster, Prashant Shenoy, Omer Rana, and Andrew A. Chien. **Serverless Computing: What It Is, and What It Is Not?** *Communications of the ACM*, **66**(9):80–92, September 2023. 11, 12

[31] Cristina Abad, Ian T. Foster, Nikolas Herbst, and Alexandru Iosup. **Serverless Computing (Dagstuhl Seminar 21201)**. Technical report, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. 12

[32] Jinfeng Wen, Zhenpeng Chen, Xin Jin, and Xuanzhe Liu. **Rise of the Planet of Serverless Computing: A Systematic Review**. *ACM Transactions on Software Engineering and Methodology*, **32**(5):1–61, September 2023. 12

[33] Mark Claypool, Shengmei Liu, Atsuo Kuwahara, James Scovell, Miles Gregg, Federico Galbiati, and Eren Eroglu. **Waiting to Play - Measuring Game Load Times and Their Effects on Player Quality of Experience**. In *Proceedings of the 19th International Conference on the Foundations of Digital Games*, pages 1–11, Worcester MA USA, May 2024. ACM. 17

[34] Simon Eismann, Joel Scheuner, Erwin van Eyk, Maximilian Schwinger, Johannes Grohmann, Nikolas Herbst, Cristina L. Abad, and Alexandru Iosup. **A Review of Serverless Use Cases and Their Characteristics**, January 2021. 38

[35] Jaime Dantas, Hamzeh Khazaei, and Marin Litoiu. **Application Deployment Strategies for Reducing the Cold Start Delay of AWS Lambda**. 50

[36] Anup Mohan, Harshad Sane, Kshitij Doshi, Saikrishna Edupuganti, Vadim Sukhomlinov, and Naren Nayak. **Agile Cold Starts for Scalable Serverless**. 50

[37] Parichehr Vahidinia, Bahar Farahani, and Fereidoon Shams Aliee. **Cold Start in Serverless Computing: Current Trends and Mitigation Strategies**. In *2020 International Conference on Omni-layer Intelligent Systems (COINS)*, pages 1–7, August 2020. 50

[38] Roman Moravskyi, Pavlo Pustelnyk, Mykola Morozov, and Yevheniya Levus. **Cloud-Based Distributed Approach for Procedural Terrain Generation with Enhanced Performance**. In *2023 IEEE 18th International Conference on Computer Science and Information Technologies (CSIT)*, pages 1–4, October 2023. 51

[39] Farouk Messaoudi, Adlen Ksentini, and Philippe Bertin. **Toward a Mobile Gaming Based-Computation Offloading**. In *2018 IEEE International Conference on Communications (ICC)*, pages 1–7, Kansas City, MO, May 2018. IEEE. 51

[40] Phanindra Kalva. **ADAPTIVE GAME CONTENT GENERATION IN CLOUD ENVIRONMENTS: LEVERAGING AI FOR PLAYER-CENTRIC EXPERIENCE OPTIMIZATION**. *INTERNATIONAL JOURNAL OF ADVANCED RESEARCH IN ENGINEERING AND TECHNOLOGY*, **16**(1):83–98, February 2025. 52

Appendix

7.3 Abstract and Concrete Dungeon Representation

The following shows the same dungeon in both concrete and abstract representation.

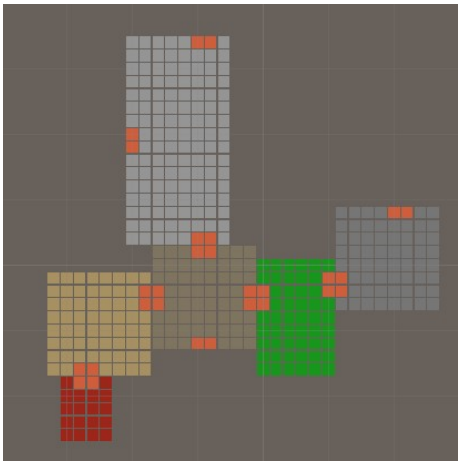


Figure 7.1: An example concrete representation of a dungeon with a main path depth of 3 and a branching depth of 1.

Listing 7.1: An example abstract representation of a dungeon with a main path depth of 3 and a branching depth of 1.

```
{
  "$id": "1",
  "partialDungeon": {
    "$id": "2",
    "dungeonRooms": [
      {
        "$id": "3",
        "anchorPos": {
          "$id": "4",
          "X": 0.0,
          "Y": 0.0
        },
```

```
      "rotation": 90,
      "Type": 0,
      "Blueprint": {
        "$id": "5",
        "id": 0,
        "topRight": {
          "$id": "6",
          "X": 8.0,
          "Y": 5.0
        },
        "doorwayBlueprints": [
          {
            "$id": "7",
            "DoorwayCells": [
              {
                "$id": "8",
                "X": 1.0,
                "Y": 5.0
              },
              {
                "$id": "9",
                "X": 2.0,
                "Y": 5.0
              }
            ]
          },
          {
            "$id": "10",
            "DoorwayCells": [
              {
                "$id": "11",
                "X": 2.0,
                "Y": 0.0
              },
              {
                "$id": "12",
                "X": 3.0,
                "Y": 0.0
              }
            ]
          }
        ]
      }
    },
    {
      "$id": "13",
      "anchorPos": {
```

```
      "$id": "14",
      "X": -1.0,
      "Y": 1.0
    },
    "rotation": 180,
    "Type": 1,
    "Blueprint": {
      "$id": "15",
      "id": 0,
      "topRight": {
        "$id": "16",
        "X": 7.0,
        "Y": 7.0
      },
      "doorwayBlueprints": [
        {
          "$id": "17",
          "DoorwayCells": [
            {
              "$id": "18",
              "X": 3.0,
              "Y": 7.0
            },
            {
              "$id": "19",
              "X": 4.0,
              "Y": 7.0
            }
          ]
        },
        {
          "$id": "20",
          "DoorwayCells": [
            {
              "$id": "21",
              "X": 7.0,
              "Y": 3.0
            },
            {
              "$id": "22",
              "X": 7.0,
              "Y": 4.0
            }
          ]
        },
        {
          "$id": "23",
```

```
            "DoorwayCells": [
              {
                "$id": "24",
                "X": 0.0,
                "Y": 3.0
              },
              {
                "$id": "25",
                "X": 0.0,
                "Y": 4.0
              }
            ]
          },
          {
            "$id": "26",
            "DoorwayCells": [
              {
                "$id": "27",
                "X": 3.0,
                "Y": 0.0
              },
              {
                "$id": "28",
                "X": 4.0,
                "Y": 0.0
              }
            ]
          }
        ]
      }
    },
    {
      "$id": "29",
      "anchorPos": {
        "$id": "30",
        "X": -16.0,
        "Y": -1.0
      },
      "rotation": 90,
      "Type": 1,
      "Blueprint": {
        "$id": "31",
        "id": 0,
        "topRight": {
          "$id": "32",
          "X": 7.0,
          "Y": 7.0
```

```
      },
      "doorwayBlueprints": [
        {
          "$id": "33",
          "DoorwayCells": [
            {
              "$id": "34",
              "X": 1.0,
              "Y": 7.0
            },
            {
              "$id": "35",
              "X": 2.0,
              "Y": 7.0
            }
          ]
        },
        {
          "$id": "36",
          "DoorwayCells": [
            {
              "$id": "37",
              "X": 7.0,
              "Y": 2.0
            },
            {
              "$id": "38",
              "X": 7.0,
              "Y": 3.0
            }
          ]
        }
      ]
    }
  },
  {
    "$id": "39",
    "anchorPos": {
      "$id": "40",
      "X": -15.0,
      "Y": -13.0
    },
    "rotation": 0,
    "Type": 3,
    "Blueprint": {
      "$id": "41",
      "id": 0,
```

```
        "topRight": {
          "$id": "42",
          "X": 3.0,
          "Y": 4.0
        },
        "doorwayBlueprints": [
          {
            "$id": "43",
            "DoorwayCells": [
              {
                "$id": "44",
                "X": 1.0,
                "Y": 4.0
              },
              {
                "$id": "45",
                "X": 2.0,
                "Y": 4.0
              }
            ]
          }
        ]
      }
    },
    {
      "$id": "46",
      "anchorPos": {
        "$id": "47",
        "X": 13.0,
        "Y": -3.0
      },
      "rotation": 270,
      "Type": 2,
      "Blueprint": {
        "$ref": "31"
      }
    },
    {
      "$id": "48",
      "anchorPos": {
        "$id": "49",
        "X": -3.0,
        "Y": 2.0
      },
      "rotation": 270,
      "Type": 2,
      "Blueprint": {
```

```
      "$id": "50",
      "id": 0,
      "topRight": {
        "$id": "51",
        "X": 15.0,
        "Y": 7.0
      },
      "doorwayBlueprints": [
        {
          "$id": "52",
          "DoorwayCells": [
            {
              "$id": "53",
              "X": 7.0,
              "Y": 7.0
            },
            {
              "$id": "54",
              "X": 8.0,
              "Y": 7.0
            }
          ]
        },
        {
          "$id": "55",
          "DoorwayCells": [
            {
              "$id": "56",
              "X": 15.0,
              "Y": 1.0
            },
            {
              "$id": "57",
              "X": 15.0,
              "Y": 2.0
            }
          ]
        },
        {
          "$id": "58",
          "DoorwayCells": [
            {
              "$id": "59",
              "X": 0.0,
              "Y": 1.0
            },
            {
```

```
                    "$id": "60",
                    "X": 0.0,
                    "Y": 2.0
                  }
                ]
              }
            ]
          }
        }
      ],
      "_numberOfRoomsOnMainPath": 3,
      "NumberOfRoomsOnMainPath": 3,
      "DesiredBranchDepth": 1
    },
    "Id": 0,
    "NumberOfRoomsOnMainPath": 3,
    "DesiredBranchingDepth": 1
}
```