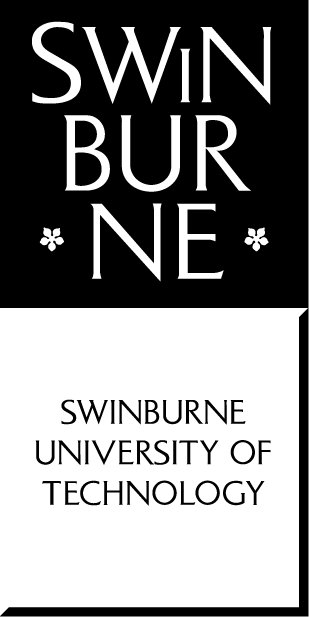
**School of Science, Computing and Engineering Technologies**

**COS10009**

**Introduction to Programming**

**Final Custom Project Report**

Project Title: Alien Shooter 2D

Student Name: Nguyen Gia Bao Pham

Student ID: 105292789

Date: 07/06/2025

**Declaration**

I declare that this report is my individual work. I have not copied from any other student’s work or from any other source except where due acknowledgment is made explicitly in the text, nor has any part of this submission been written for me by another person.

**Signature: Nguyen Gia Bao Pham**

Table of Contents

[Part A: Introduction 1](#_Toc200218488)

[Introduction to my project report 1](#_Toc200218489)

[Part B: Literature Review 2](#_Toc200218490)

[Introduction 2](#_Toc200218491)

[Custom Project 2](#_Toc200218492)

[Conclusion 3](#_Toc200218493)

[Part C: Methods 4](#_Toc200218494)

[Game logic structure chart 4](#_Toc200218495)

[Game Initialization and Scene Control 4](#_Toc200218496)

[Screne Management 4](#_Toc200218497)

[Player Movement and Drawing 5](#_Toc200218498)

[Enemies (Alienships) and Bosses 5](#_Toc200218499)

[Shooting Mechanics and Bullets 5](#_Toc200218500)

[Hearts, Coins and Power-up 5](#_Toc200218501)

[Boss Laser and Combat 5](#_Toc200218502)

[Explosion and Visual Feedback 6](#_Toc200218503)

[Game Start and Difficulty Buttons 6](#_Toc200218504)

[Sound Effects and Music 6](#_Toc200218505)

[Image Resources and Editing 6](#_Toc200218506)

[What different between my custom game with other games 6](#_Toc200218507)

[Part D: Alien Shooter 2D 7](#_Toc200218508)

[Game Walkthrough 7](#_Toc200218509)

[Start Menu with Difficulty Selection 8](#_Toc200218510)

[Player Movement and Shooting 8](#_Toc200218511)

[Alienships and Boss Mechanics 8](#_Toc200218512)

[Animations and Sounds 8](#_Toc200218513)

[Advantage Features 9](#_Toc200218514)

[Part E: Refenrences 12](#_Toc200218515)

## 

## Part A: Introduction

### Introduction to my project report

For purposose aiming HD band, so I come up with my custom project. The following paragraphs is my report talks about my custom game called Alien Shooter 2D, which is a 2D space shooter game made with Ruby and the Gosu library. The game is inspired by the classic space invader games where you control a spaceship to fight against alien ships and a boss enemy. There are three certains level, you can choose one of three difficulty levels, and the game has enemies that move in different ways, items like coins and health hearts you can collect, and cool sounds and animations to make it more fun.

In the game, you need to control your spaceship for move your spaceship avoid the alienships (U.F.O) and bosses (which is the alien), shoot bullets that change depending on your score and difficulty, and try to survive as long as you can. The enemies spawn randomly and get harder over time. There’s also a start menu and an ending screen that tells you how well you did.

This project shows how I understanding about Ruby languge and shows how I used Ruby’s object-oriented programming to create different game objects, how I handled animations using sprite sheets, and how I made the game react to player actions. Overall, Alien Shooter 2D is a fun and challenging game that brings together graphics, sound, and gameplay mechanics in a simple way with Ruby code and friendly for beginner in Ruby.

## Part B: Literature Review

### Introduction

This game project, Alien Shooter 2D, was build using the Ruby Gosu library. It’s a classic 2D arcade shooter where the player control a spaceship and try to survive waves of alienships, bosses, and bullets. The goal of this review is to explore how similar 2D games are designed in the literature and how theories around player feedback, difficulty progression, and game engagement apply. While most game dev resources talk about Unity or Unreal, not many talk about lightweight game frameworks like Gosu or using Ruby language for games. So this review focuses more on how core mechanics like game loop, collisions, animation, and sound work together in a simple but still immersive shooter experience.

### Custom Project

**Gosu Framework and Game Structure**

Gosu is a 2D game library that give developers control over rendering, sounds, and input. It's small and fast, good for making retro-style games. Pilgrim (2021) said Gosu is really suitable for games that need precise sprite control but not heavy 3D processing. In Alien Shooter 2D, the main game is wrapped inside a class that inherits from Gosu::Window, which control everything drawing, updating, handling keys and mouse.

Instead of putting all logic in one place, the program use separate classes like Player, Bullet, Alienship, and so on, which is a good example of object-oriented design. This keep the game code clean and let feature like collision or animation update be reused easily.

**Real-Time Feedback and Flow Theory**

According to Csikszentmihalyi’s (1990) flow theory, players enjoy a game the most when it balance between challenge and skill. If it’s too easy, people get bored or too hard, they give up. This game actually try to achieve flow by slowly increasing enemy spawn rate and adding more bullets when player score go up. This sort of reward loop helps players stay motivated. The triple and quadruple bullet upgrades at 6000 and 16000 points make players feel they’re getting better and stronger, which is a part of flow experience.

Also, feedback is key. Every time the player shoot or get hit, there’s a sound, some explosion or screen shake. This type of multisensory feedback help keep attention and engagement (Sweetser & Wyeth, 2005). Even small thing like shaking the screen when hit makes it feel more intense and real.

**Input Latency and Responsiveness**

One of the thing that really affect game feel is input latency — how fast the game react after a player press a key or click. While Gosu doesn’t have a built-in way to test latency, it’s known for being lightweight and responsive (Gosu Docs, 2023). In Alien Shooter 2D, input handling is done with simple checks like Gosu.button\_down? Gosu::KB\_LEFT, so the player movement is smooth and immediate. This fast reaction time is important especially in action games where delay make player feel out of control.

**Collision and Difficulty Scaling**

The game use distance-based collision detection (via Gosu.distance) which is simple but work well for 2D sprite-based games. It check the space between the center of two objects, and if it’s small enough, a hit is detected. This is used for bullets hitting aliens, player collecting coins, and boss lasers hitting player.

To make the game more interesting, the difficulty scale not just in speed or spawn rate, but also in how the player weapons evolve. It match what Gee (2003) call “pleasantly frustrating”—a condition where player is constantly challenged but still feel they can overcome it.

**Sounds, Animations, and Game Feel**

Visuals and sound also play a huge role in how polished the game feels. There are separate image tiles for explosions, blood, and animated sprites, and the game use animation frames based on milliseconds. That give a feeling of movement without needing complex physics. According to Isbister (2016), games that deliver feedback through multiple senses increase player satisfaction and immersion.

Even though it’s a small game, it include custom sound effects for actions like shooting or collecting coins. Those small touches really make a big difference in how the game is perceived.

### Conclusion

This game may be built in a simple Ruby framework, but it demonstrate many key principles from game development and psychology literature. It manage to combine responsive controls, progressive difficulty, real-time feedback and animated effects to create a surprisingly engaging experience. The use of flow theory, collision detection, and reward systems are aligned with practices discussed in both academic and indie game dev sources.

Future development could include saving high scores, adding multiplayer, or using machine learning to make bosses smarter. For now, Alien Shooter 2D show that with the right design choices, even a lightweight game can feel fun, intense, and rewarding.

## Part C: Methods

### Game logic structure chart



### Game Initialization and Scene Control

In my game, the initialize method set up the game window (1000x800 pixels), loads images and music, defines initial variables (like @scene, @font, etc.), and plays background music for the start screen. It also defines color choices for UI buttons and prepares the screen shake effect (@shake\_timer, @shake\_offsets). This setup creates an immersive entry point for players.

Moreover, the initialize also define the enemy rate and three different mode play for player to select.

### Screne Management

The @scene variable controls whether the game is showing the start menu, gameplay, or end screen. The methods update, draw, and button\_down use this variable to call the appropriate logic based on what the player is doing. This modular scene handling keeps the game flow organized and easy to manage.

### Player Movement and Drawing

Functions like move\_left, move\_right, move\_up, and move\_down are used to move the player's spaceship while ensuring it stays within the screen boundaries. The draw\_player method animates the spaceship using a tileset image, drawing one frame at a time based on elapsed milliseconds. The player can freely move in 2D space using arrow keys, enhancing interactivity.

### Enemies (Alienships) and Bosses

The Alienship and Boss classes define enemy behavior. Enemies are randomly spawned and move with velocity (vel\_x, vel\_y). The move\_ufo and move\_boss functions update their positions. The draw\_ufo and draw\_boss methods render animated sprites based on preloaded tilesets. This gives life to the antagonists of the game.

### Shooting Mechanics and Bullets

The Bullet class defines bullets fired by the player, and shoot updates their Y-position as they move. Bullet generation logic in button\_down\_game varies based on score and difficulty. For example, the player shoots more bullets as their score increases, creating an engaging skill-reward loop. The bullets are drawn with draw\_bullet.

### Hearts, Coins and Power-up

The Heart, Blood, and Coin classes define collectable items or effects. Hearts increase player lives, coins boost score, and blood provides visual feedback. These objects fall from the top and interact with the player upon collision, with sounds played on successful collection or hit. Visual effects like blood or explosions are animated using tilesets, handled by draw\_heart, draw\_coin, and draw\_blood.  
  
For implement the power-up mechanic, I used conditional if statements to increase the number of bullets shooted by the player's spaceship based on their score. As the player collects coins and accumulates points, the ship gains additional firepower. For instance, in hard mode, reaching 1,000 points upgrades the ship to fire two bullets per shot, while reaching 6,000 points enables triple bullets. In insane mode, the thresholds are higher: the player unlocks triple bullets at 8,000 points and four bullets at 16,000 points.

### Boss Laser and Combat

Bosslaser class represents bullets shot by bosses. Like bullets, they fall from the top and are drawn with draw\_bosslaser. If they collide with the player, the player's health decreases and screen shake is triggered to provide visual feedback (@shake\_timer, @shake\_offsets). This adds difficulty and tension during boss fights.

### Explosion and Visual Feedback

When enemies or bosses are hit, Explosion objects are created and animated with draw\_explosion. These animations enhance the visual satisfaction of combat. The game uses logic to remove explosions once the animation ends (remove\_explosions), keeping memory use efficient. On the other hand, your space ship also display explosion when you hit the bosses, boss lazer or even alienships. After you hit, explosion audio play and your heart decrease.

### Game Start and Difficulty Buttons

When the game start, it show a cool menu screen where player can choose between easy, hard, or insane mode. These button glow blue when you move the mouse on top of them, which make it easier to see which one you selecting. Each mode change how fast enemies come out and what kind of bullet the player can use later. The game remember your choice using something called @difficulty. Colored boxes also help show the buttons better so it feel more like a real game UI.

### Sound Effects and Music

The game have nice sound that make it more fun. It use background music and also sound when you shoot or when a spaceship explode. There’s even a sound when you collect coin or get hit. All the music and sound effect was found online. The music come from [Pixabay](https://pixabay.com/music/search/diamonds%20drop/) and the sound effect from [OpenGameArt](https://opengameart.org/content/8-heals-and-buffs-sfx). The game use .play to make the sounds happen, and that really help make the action more exciting.

### Image Resources and Editing

The alien ships, player ships, coins, and even the background picture all come from online too. A lot of them were found on [Pinterest](https://www.pinterest.com/pin/14988611254318242/), and some started as GIFs that I have to turn into PNG so the game can use them. I use [FreeConvert](https://www.freeconvert.com/gif-to-png) for that. To make everything fit better in the game, I also resized them using [Fotor](https://www.fotor.com/vi/features/resize/). The images get loaded in the code with Gosu::Image.new or Gosu::Image.load\_tiles for animated one.

When the player lose all their hearts (or lives), or when they survive through all the enemies, the game show an end screen. It say if you survived or not, and show your score too. The game will ask you for play again, you can press P to start over, or press Q to leave the game. This part of the game is handle by the initialize\_end method and the draw\_end screen. It’s simple but helpful so you know how you did.

### What different between my custom game with other games

My game Alien Shooter 2D stand out when compare to other alien shooter game with the same idea. Most of them just give you a basic ship, you move around, shoot alien, and that’s pretty much it—same style, same weapon, not much changing. But I want my game to feel more alive and exciting, so I add stuff like power-up when you collect coins. For example, when you get enough points, your ship doesn’t just shoot one bullet, it start firing double, triple, or even four bullets at once depend on how much score you got and the difficulty you pick. That make player feel more powerful and actually want to keep playin to get better firepower.

I also add hearts that can heal your ship when it get hit and coins that boost your score. These small things add more reason to move around and take risk, instead of just avoid enemy. Plus, the enemies not always same. There’s alienship, boss, and boss laser all come at random time, and each one move different way, so you always have to react and think. I think a lot of other game don’t really do this—they just throw same type of enemies over and over.

Another thing is the mode play. There are three certains mode play for player included easy, hard, and insane really do feel different in my game. In insane mode, you get more enemies, faster attacks, and need higher score to unlock better shooting. It’s not just a label like in other game—it actually change how you play. I also use different sounds and animation for each action, like when you shoot, hit, collect coins, or get hurt. That give it more feeling and make the game more fun and not so dry like others I played.

Moreover, one different of my game from other game is my game allow for player puts their name into my gama and I can use a online PostgreSQL database from Railway to save the score after each time the game end. When the player ship got destroy, the game take their name, the score they earn, what difficulty they pick, and the current time. Then it put all that into a table call scores in the database. I use sequel gem to connect Ruby with the database, and the link is store safely in a .env file using dotenv. This help make sure my data get send to the right place and stay private. With this setup, I can keep all player history and maybe later make a website to show top scores. This also a good way for beginner in Ruby to learn how to save game data online, not just on the screen.

In a nutshell, I try to make my game not only about shooting but also about feeling progress, reward, and challenge, which I don’t see much in many other alien shooting game.

## Part D: Alien Shooter 2D

### Game Walkthrough

My game Alien Shooter is a 2D top-down shooter where the player can control a spaceship using the arrow keys and fire enemies with the spacebar. At the beginning, it start at a menu screen with fun background music and some cool animation. You can choose between easy, hard, or insane mode by clicking buttons, and they even change color when you hover the mouse over them. When the game start, enemy ships begin falling down from the top of the screen, and the player have to move left and right (or up and down too) to avoid them or shoot bullets to destroy them. You get points by shooting enemies and collecting coins. Sometimes bosses show up—they're bigger and shoot back with laser. If the player lose all their health or survive a ton of enemies, it go to the end screen where you can either press P to restart or Q to quit. After you finish play, your data of your name, score, difficulty of play mode and display time play.

### Start Menu with Difficulty Selection

The Alien Shooter 2D game use a variable called @scene to switch between the menu and the gameplay. When the game is in the start scene, you see buttons on screen for different difficulty levels. The color of the buttons changes when the mouse move over them, which make it feel interactive. When you click a button, the game sets the enemy spawn rate based on your choice—so in insane mode, enemies appear faster. This setup is good for beginners because it teaches how to track game state using simple logic.

### Player Movement and Shooting

The player must control the ship and move it around using the arrow keys. The ship doesn’t move too fast, just enough to dodge bullets and chase coins. Pressing the spacebar fires a bullet upward, and as your score gets higher, your ship shoot more bullets at once. The player’s data—like position, score, and lives—is stored in an object made from a Player class. This make it easy for beginners to understand how to store and use information using object and class.

### Alienships and Boss Mechanics

Alienships randomly spawn from the top of the screen and float down in different directions. Some go straight down, others wiggle side to side. Bosses appear sometimes and they are tougher—they have HP and shoot red laser bullets at the player. To check if something hit you, the game use a simple function from Gosu to measure the distance between two objects. If the distance is small, then it’s a collision. This way is simple and work great for learning collision detection.

### Animations and Sounds

Everything in the game moves with animation. For example, the ship's fire, the coins spinning, and the explosion all use a series of image frames (sprites) loaded from a GIF file that was converted and resized. The code use Gosu::Image.load\_tiles to load these image as frames to animate. Sound effects are added too—when you shoot, explode, or collect something, it make sound. This is done using Gosu::Sample.new and Gosu::Song.new. These media files were all found from free resource websites like Pixabay, OpenGameArt, and Pinterest, then converted using tools like FreeConvert and resized in Fotor.

### Advantage Features

**Screen Shake Effect**

A fun feature is the screen shake slightly when you get hit by a boss or alienship or enermy bullets. It’s not just a visual trick—it make the game feel more intense. It’s done by randomly shifting the background and elements a few pixels for a short time. This is controlled using a timer (@shake\_timer) and random numbers. It’s a small thing, but it make a huge difference in how the game feels and make more excited in game experience for the player.

**Object-Oriented Design**

Each part of the game like the player, bullet, or boss is a separate class. That’s the basic of object-oriented programming (OOP). It help me to understand how to group code into things (object) that have their own behavior. That realy help me for improve the process coding this project and easier foe define the bug if it not working for something else happen when the code work.

**Clear Update and Draw Loops**

There’s a clear separation in the game between the update methods (which handle logic like movement and collisions) and the draw methods (which handle displaying things on screen). This show new Ruby users how to structure their game loop cleanly.

**Gosu Basics**

By using the Gosu gem, the game teach new users how to play sound, animate sprites, detect keyboard input, and display text. It’s not too much at once, just enough to get started making something real.

**Modular Methods**

My code demonstrate to break things down into small pieces like move\_player, draw\_bullet, and shoot\_boss. This make it easier for anyone who read the code and understand it clearly and follow or find mistake when something not work.

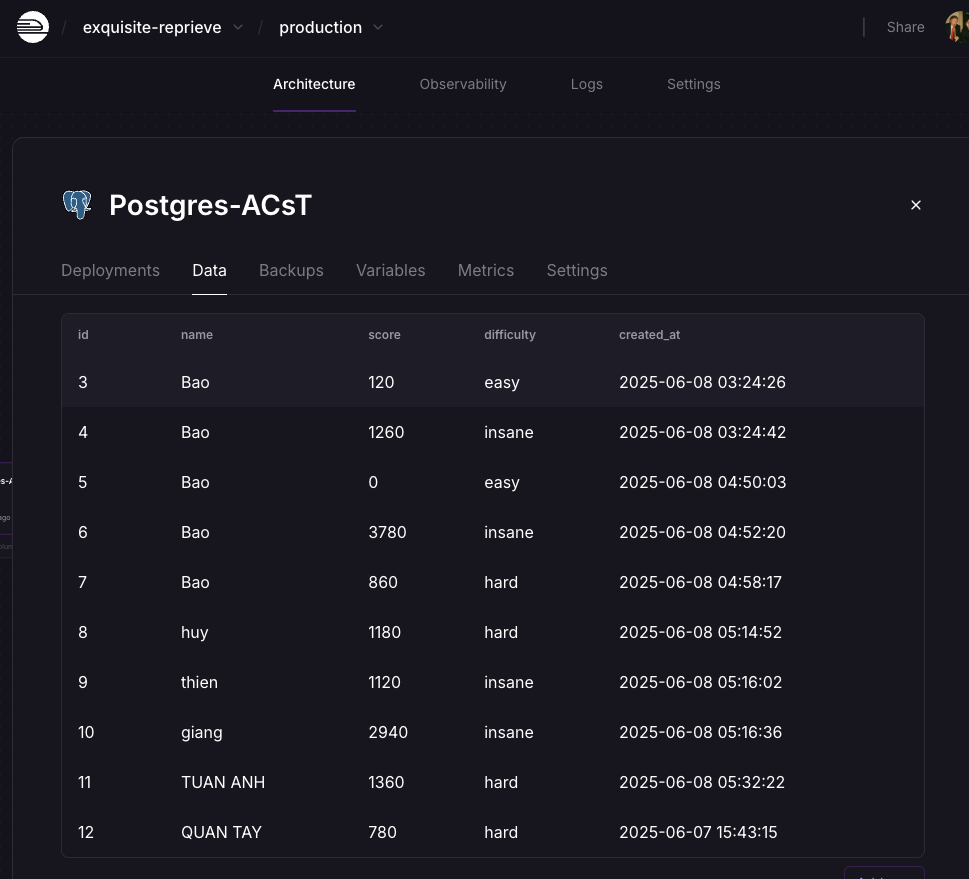
**Random Elements**

Enemies like alienships and bosses appear randomly using rand. For me, as a beginner and it show how randomness work and why it’s fun to use in game design.

**Sound and Asset Management**

The game show how to load files, play music, and use different image files. It’s a basic way to learn how to connect your code with the outside resource (like sound and pic) without needing complex code.

**Database**PostgreSQL database host on Railway to keep track of player score. PostgreSQL is a powerful database system that can store table of data, like name, score, difficulty, and time when the game end online (PostgreSQL Global Development Group, 2024). Railway is a cloud platform that make it easy to create and connect to the database without setting up server by yourself (Railway, n.d.). In the game, when player finish playing, the data get send to the database using the sequel gem, and the URL for connecting is hide inside a .env file to keep it safe. This setup help my game save scores online, so I can look back at old scores or build a leaderboard later. It also show how real game can use online database to make data not lost after closing the game.

****

**( Screenshot of the PostgreSQL database )**

**Input player name**

At the begin of the game, player can type their name or their ingame nickname before get into a windown for choose the difficulty. I make a simple name input screen where user can type letters, number, symbol and even space if they want name with two word. The name can be lowercase or uppercase depend on how player press the key. I use Gosu to catch keyboard press, so when you type A, 3, or @, it just add into the name. If player press Enter and the name not empty, the game go to the main menu. This make the game feel more personal and also play importance rol for define the player on database and save the player name with they score into the database.

**Link my game to database**

To connect my game with an online database, I use PostgreSQL hosted on Railway. In my Ruby code, I use the sequel gem which store in file score.rb included code help with database connection and queries, and I load the Railway database link securely using the dotenv gem from a hidden .env file. This means the game know how to connect to the database without exposing the real URL in public code. I create a Score class that link to the scores table, which stores the player's name, score, difficulty level, and the time they finished the game. If the table not already exist, the code will make it. When the game ends, it automatically send the score to the Railway database, so I can check later or use it for leaderboard. This setup make the game more dynamic and realistic, because data is saved online and won’t disappear when the game close (Fitzgerald, 2020; Sequel, n.d.; Railway, n.d.). I also use a git file for sensitive data protection purpose.

## Part E: Refenrences

* Csikszentmihalyi, M. (1990). *Flow: The psychology of optimal experience*. Harper & Row.
* Gee, J. P. (2003). *What video games have to teach us about learning and literacy*. Palgrave Macmillan.
* Gosu Documentation. (2023). *Gosu Game Development Library*. <https://www.libgosu.org>
* Isbister, K. (2016). *How games move us: Emotion by design*. MIT Press.
* Pilgrim, M. (2021). *Beginning Game Programming with Ruby and Gosu*. GameDev Press.
* Sweetser, P., & Wyeth, P. (2005). GameFlow: A model for evaluating player enjoyment in games. *Computers in Entertainment*, *3*(3), 3–3. https://doi.org/10.1145/1077246.1077253
* Claypool, M., & Claypool, K. (2006). Latency and player actions in online games. Communications of the ACM, 49(11), 40–45. https://doi.org/10.1145/1167838.1167860
* Csikszentmihalyi, M. (1990). Flow: The Psychology of Optimal Experience. Harper & Row.
* Pagulayan, R. J., Keeker, K., Wixon, D., Romero, R. L., & Fuller, T. (2003). User-centered design in games. In The human-computer interaction handbook (pp. 883–906). CRC Press.
* Pixabay. (n.d.). Royalty-free music. Retrieved from <https://pixabay.com/music/search/diamonds%20drop/>
* OpenGameArt. (n.d.). Heals and buffs sound effects. Retrieved from <https://opengameart.org/content/8-heals-and-buffs-sfx>
* Pinterest. (n.d.). Alien spaceship game art. Retrieved from <https://www.pinterest.com/pin/14988611254318242/>
* FreeConvert. (n.d.). GIF to PNG Converter. Retrieved from <https://www.freeconvert.com/gif-to-png>
* Fotor. (n.d.). Resize image. Retrieved from <https://www.fotor.com/vi/features/resize/>
* PostgreSQL Global Development Group. (2024). PostgreSQL documentation. <https://www.postgresql.org/docs/>
* Railway. (n.d.). Railway – Deploy faster. <https://railway.app/>
* Sequel. (n.d.). Sequel: The Database Toolkit for Ruby. https://sequel.jeremyevans.net/
* Fitzgerald, M. (2020). Learning Sequel: A Ruby ORM for SQL Databases. O’Reilly Media.  
  Railway. (n.d.). Ship faster with modern developer infrastructure. <https://railway.app/>