Arquitectura de Solución: Diseño y Lineamientos Técnicos para Power Garden

Santiago Botero García

Laura Natalia Perilla Quintero

José David Castillo Rodríguez

Escuela Colombiana de Ingeniería Julio Garavito

ARSW-1: Arquitectura de Software

Diego Andrés Triviño Gonzales

Diciembre 03, 2025

Contenido

Justificación Técnica

La arquitectura de Power Garden ha sido diseñada para cumplir simultáneamente con los objetivos técnicos, de negocio y académicos del proyecto. Cada decisión tecnológica responde a criterios de eficiencia, escalabilidad, mantenibilidad y alineación con las necesidades de un juego multijugador en tiempo real, accesible desde navegador y dispositivos de gama baja. El stack seleccionado: Unity como motor de juego, Svelte para la interfaz, y un back-end en ASP.NET responsable de la persistencia con MongoDB y autenticación JWT. Conforma una arquitectura moderna, ligera y altamente mantenible. Cada componente cumple un rol claro, evitando complejidad innecesaria y privilegiando la escalabilidad, la velocidad de desarrollo y la alineación con las necesidades reales del proyecto.

Motor de Juego: Unity

Unity es el núcleo de la experiencia de Power Garden. Su compatibilidad con WebGL permite entregar el juego directamente en navegadores sin instalación, manteniendo al mismo tiempo portabilidad hacia PC, móviles y consolas desde una sola base de código. Unity fue seleccionado como núcleo del sistema debido a su capacidad para:

- **Portabilidad y compatibilidad WebGL: Permite** que el juego se ejecute directamente en navegadores, sin instalaciones adicionales, y mantiene la opción de portar a PC, móviles y consolas desde la misma base de código. Esto es crítico para el objetivo de accesibilidad y penetración en mercados con hardware limitado.
- **Multijugador en tiempo real con Relay:** La integración con Unity Relay evita la necesidad de administrar servidores propios para networking, gestionando NAT traversal, sesiones estables y comunicación segura entre jugadores. Esto reduce costos operativos, riesgos y tiempo de implementación, alineándose con la meta de un ROI positivo y un time-to-market rápido.
- **Ecosistema consolidado:** Unity cuenta con una comunidad global, documentación extensa y soporte para herramientas avanzadas, que permiten agregar un valor diferencial aumentando la inmersión de los jugadores.

En síntesis, Unity permite cumplir los requisitos funcionales de concurrencia y latencia en tiempo real, mientras se mantiene un desarrollo ágil y escalable.

Front-End: Svelte

Svelte se eligió para la capa de interfaz debido a su enfoque compiler-first, queelimina la sobrecarga típica en tiempo de ejecución, generando bundles extremadamente pequeños y un rendimiento superior en navegadores. Esto es especialmente crítico en aplicaciones donde la respuesta inmediata impacta directamente la experiencia del usuario, como paneles de control, pantallas de matchmaking o interfaces integradas con el juego. Esto es coherente con:

- La necesidad de optimización para navegadores y hardware limitado, donde cada milisegundo de latencia afecta la experiencia del usuario.
- La eficiencia en el manejo del estado y la reactividad mediante Runes, que reduce el código repetitivo y acelera ciclos de desarrollo.
- La facilidad de integración con ASP.NET y servicios externos.

Svelte ofrece actualizaciones fluidas y predecibles sin el costo de un DOM virtual. La reducción de código repetitivo y la simplicidad del manejo de estado incrementan la productividad del equipo y acortan los ciclos de desarrollo. Empresas como IKEA y The New York Times ya han demostrado su solidez en producción.

Back-End: ASP.NET Core

El back-end está construido sobre ASP.NET Core, limitado deliberadamente a las responsabilidades esenciales del proyecto:

- Persistencia de datos
- Exposición de endpoints API
- Autenticación y autorización con JWT

ASP.NET Core ofrece uno de los mejores rendimientos entre los frameworks modernos, incluso en escenarios de alta concurrencia. Su madurez y predictibilidad lo convierten en una opción ideal para un backend que, aunque ligero, debe ser confiable.

La autenticación mediante JSON Web Tokens facilita la integración tanto con Svelte en el front-end como con el cliente de Unity, garantizando sesiones seguras sin necesidad de gestionar cookies o estados de servidor.

Esta arquitectura mínima reduce costos operativos y evita la complejidad de servidores en tiempo real, ya que esa responsabilidad recae en Unity Relay. El resultado es un backend robusto pero simple, fácil de mantener y con un ecosistema estable.

Capa de Datos: MongoDB

La arquitectura de datos se simplifica utilizando exclusivamente MongoDB Atlas, alojado en un clúster administrado en AWS (región US). Esta decisión elimina la necesidad de tecnologías adicionales, reduciendo la complejidad operativa y manteniendo un modelo de persistencia robusto, flexible y altamente escalable.

MongoDB, con su enfoque orientado a documentos, permite modelar de manera natural elementos fundamentales del juego como perfiles de usuario, progresión, estadísticas, historial de partidas y compras. Su esquema flexible favorece la evolución continua del producto sin migraciones rígidas, lo que es especialmente valioso en entornos iterativos como el desarrollo de videojuegos. Aunque no está diseñado para reemplazar un motor transitorio especializado, MongoDB Atlas ofrece:

- Baja latencia gracias a su infraestructura en AWS
- Transacciones multi-documento para garantizar integridad cuando es necesario
- Índices avanzados y pipelines de agregación para consultas eficientes
- Autoscaling en almacenamiento y rendimiento
- Backups automáticos y alta disponibilidad fuera de la caja

La elección de Atlas garantiza una operación simplificada: la plataforma gestiona replicación, seguridad, monitoreo, balanceo y actualizaciones automáticas, permitiendo que el equipo se enfoque en la lógica del juego y no en administrar bases de datos. En conjunto, MongoDB Atlas proporciona una solución de persistencia moderna, confiable y preparada para escalar conforme crezca la base de jugadores.

Encaje Operativo y Eficiencia de Costos

La arquitectura propuesta se caracteriza por ser ligera, eficiente y escalable, permitiendo un crecimiento progresivo sin incurrir en costos innecesarios desde las primeras etapas del proyecto. En la capa de aplicación, el backend en ASP.NET se ejecuta en contenedores, lo que habilita despliegues consistentes y un uso óptimo de recursos. Más adelante se describe en detalle cómo esta aplicación se ejecuta en Azure, utilizando un enfoque rentable que incluye un Load Balancer, dos máquinas virtuales y un clúster de Docker Swarm orquestado con Traefik, permitiendo manejar réplicas, balanceo interno y rutas seguras con una inversión controlada. La segmentación mediante VNETs complementa este diseño con medidas de seguridad claras y bien delimitadas sin elevar la complejidad.

En el lado del cliente, Unity con WebGL y Relay elimina la necesidad de servidores dedicados para networking en tiempo real, reduciendo enormemente el costo operativo asociado al multijugador. Por su parte, el front-end en Svelte minimiza el consumo de recursos gracias a su modelo compiler-first, lo que se traduce en un uso más eficiente del hosting donde se despliegue.

En conjunto, el stack no solo es técnicamente sólido, sino también financieramente responsable, permitiendo avanzar desde el prototipo hasta una operación estable y escalable con una progresión de costos predecible y controlada. Este encaje operativo garantiza que la inversión se dirija únicamente a los componentes que agregan valor directo, manteniendo el enfoque en la evolución del producto y no en la infraestructura.

Relación con los Objetivos de Negocio

El business case original establece como metas:

- Accesibilidad tecnológica para jugadores con hardware limitado.
- Experiencia multijugador en tiempo real, competitiva y envolvente.
- Modelo de monetización escalable con bajo CapEx y OpEx.

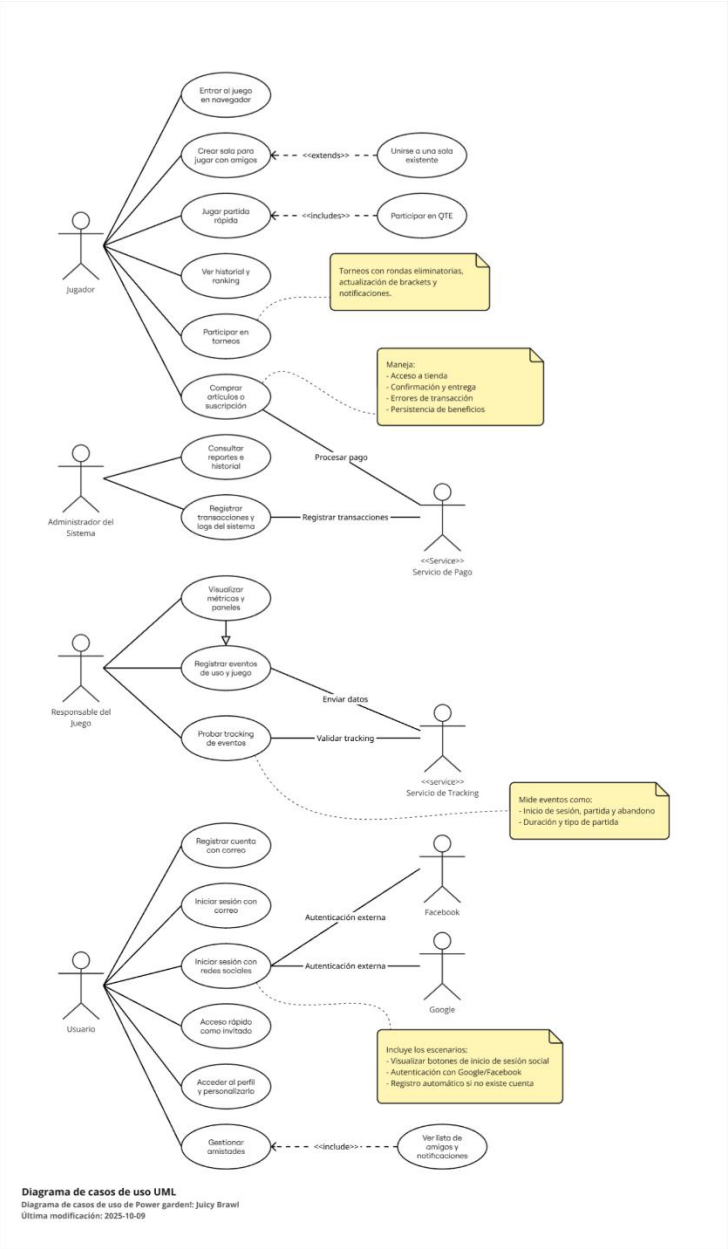
Cada tecnología seleccionada respalda directamente estos objetivos:

Objetivo de negocio	Tecnología elegida	Justificación técnica
Accesibilidad y compatibilidad web	Unity WebGL + Svelte	Permite ejecutar el juego en navegadores sin instalación y UI ligera
Multijugador en tiempo real	Unity Relay + ASP.NET	Relay maneja networking; ASP.NET gestiona autenticación y endpoints
Baja latencia y eficiencia	Svelte + MongoDB Atlas	Bundles pequeños y consultas rápidas reducen la latencia del sistema
Escalabilidad y ROI controlado	Contenedores, Azure, MongoDB Atlas	Infraestructura escalable y costos operativos predecibles

Diagramas de Arquitectura

Figura 1

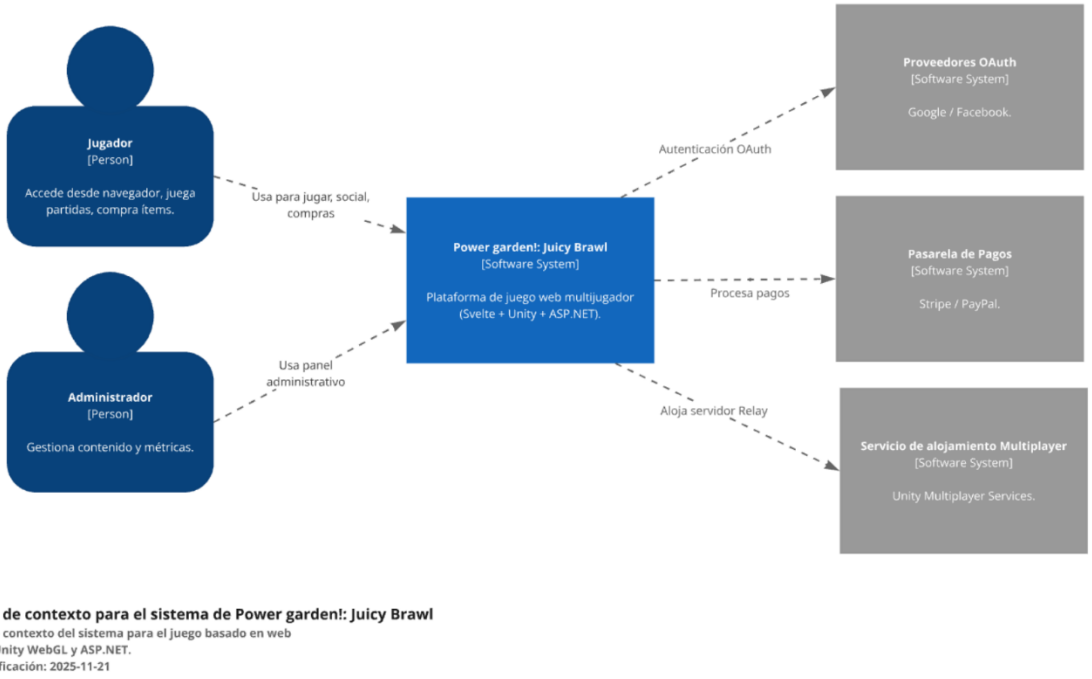
Diagrama de casos de uso UML



Nota. Los usuarios primarios de la aplicación corresponden a jugadores con credenciales autorizadas para autenticarse y acceder a la funcionalidad del juego. Adicionalmente, los administradores del sistema son responsables de monitorear, analizar y gestionar las métricas y el tracking interno del comportamiento dentro de la aplicación.

Figura 2

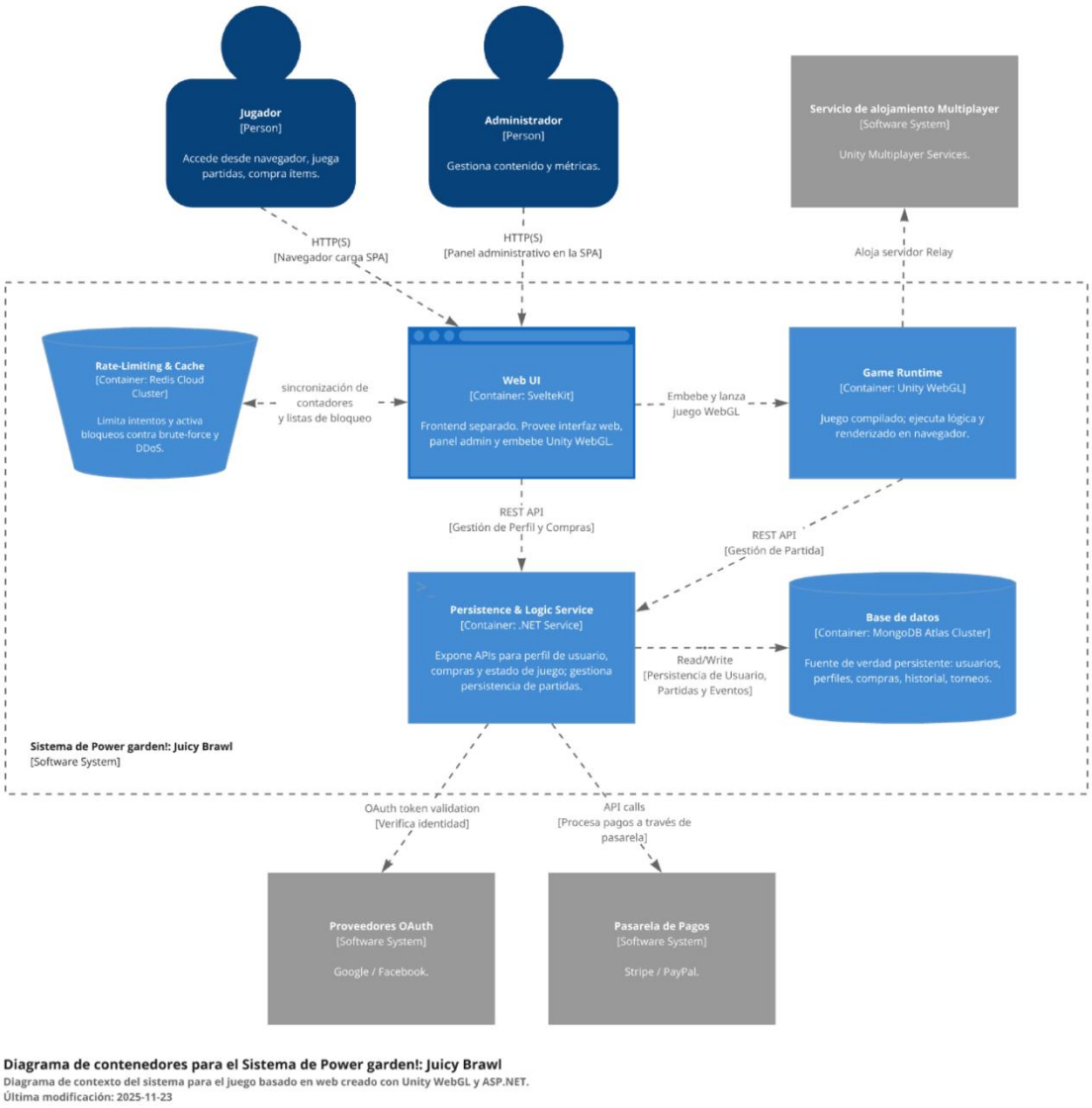
Diagrama de contexto C4 para el sistema de Power Garden: Juicy Brawl!



Nota. La consolidación de las arquitecturas AS-IS y TO-BE permite la identificación de actores externos, específicamente proveedores de autenticación y pasarelas de pago, para su integración en el ecosistema del sistema.

Figura 3

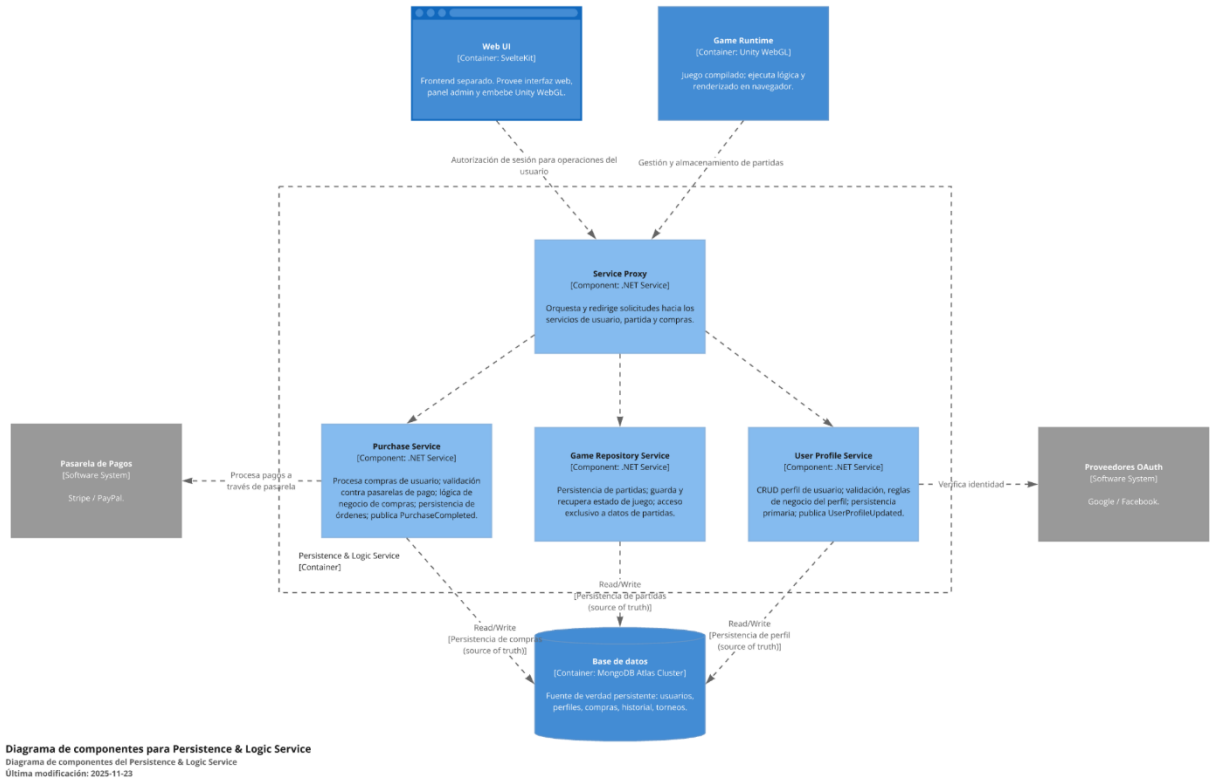
Diagrama de contenedores C4 para el Sistema de Power Garden: Juicy Brawl!



Nota. Diagrama de contenedores que representa la interacción entre el front-end, el back-end y el motor de juego, integrando MongoDB para persistencia de datos y Redis para throttling y control de rate limiting, bajo un enfoque TO-BE.

Figura 4

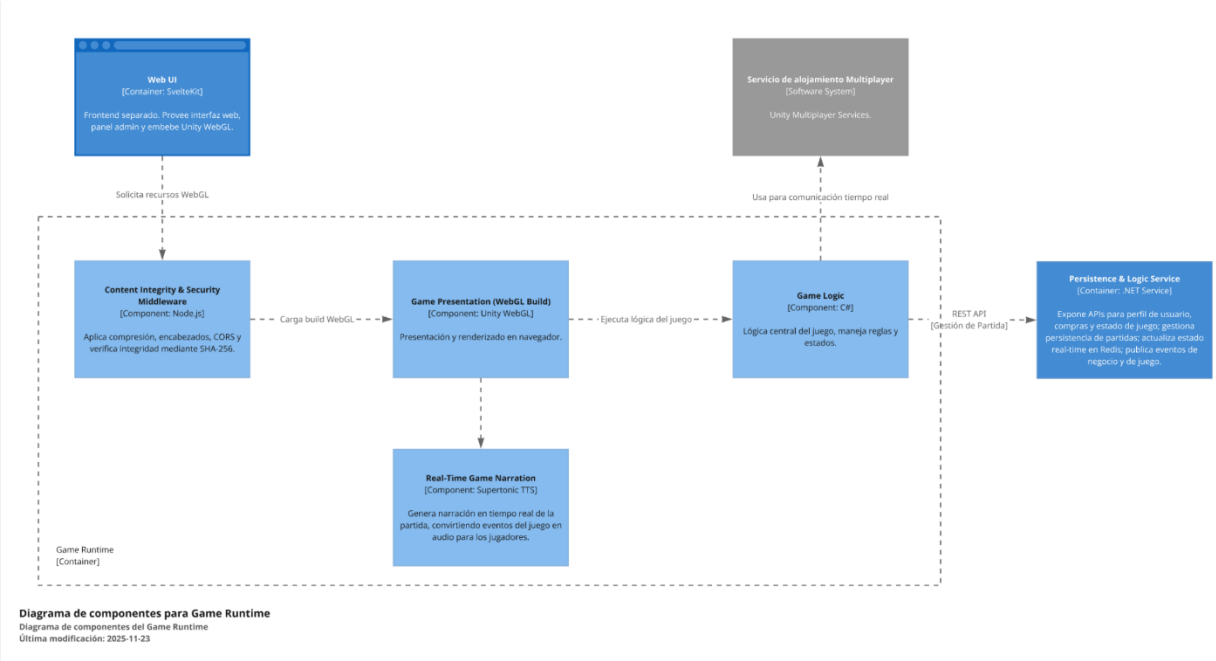
Diagrama de componentes C4 para el servicio de Persistencia y Logica



Nota. Diagrama de componentes que describe exclusivamente la orquestación del back-end con el servicio proxy.

Figura 5

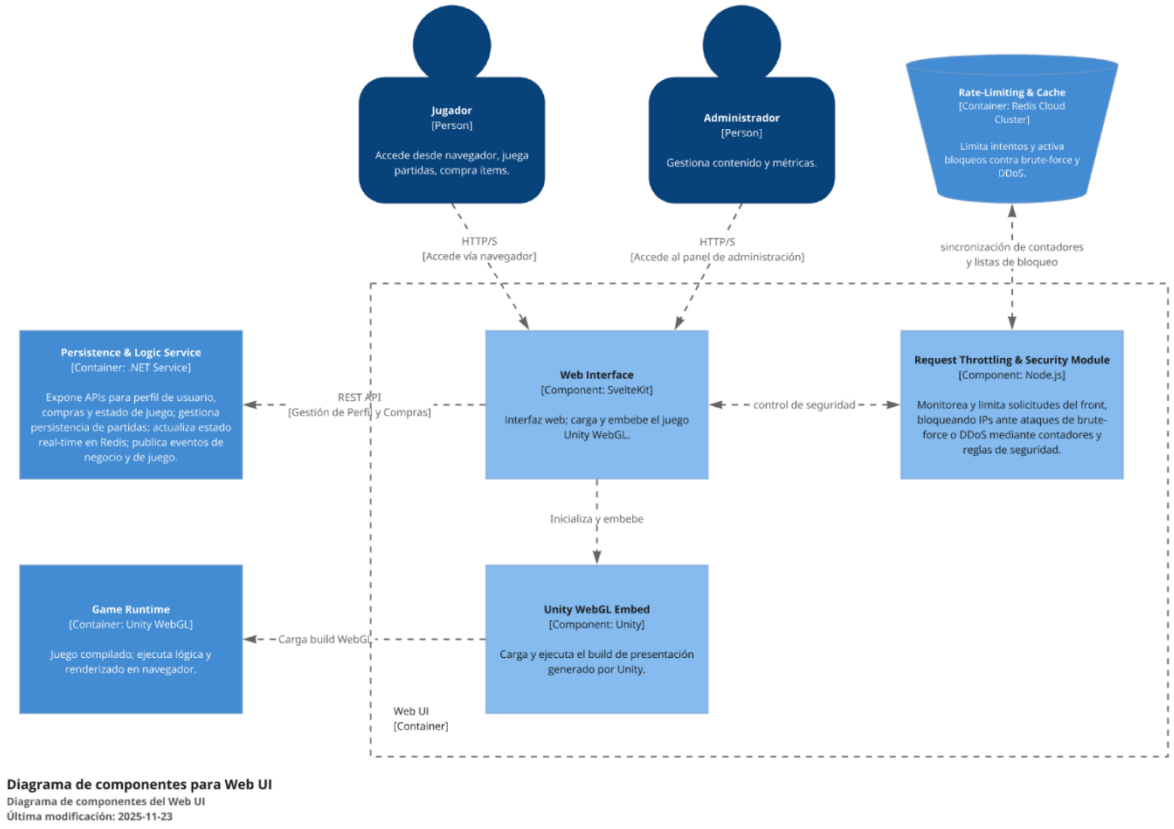
Diagrama de componentes C4 para el Game Router



Nota. Diagrama de componentes que representa el funcionamiento del juego WebGL, integrando Relay como servicio de alojamiento para la funcionalidad multiplayer.

Figura 6

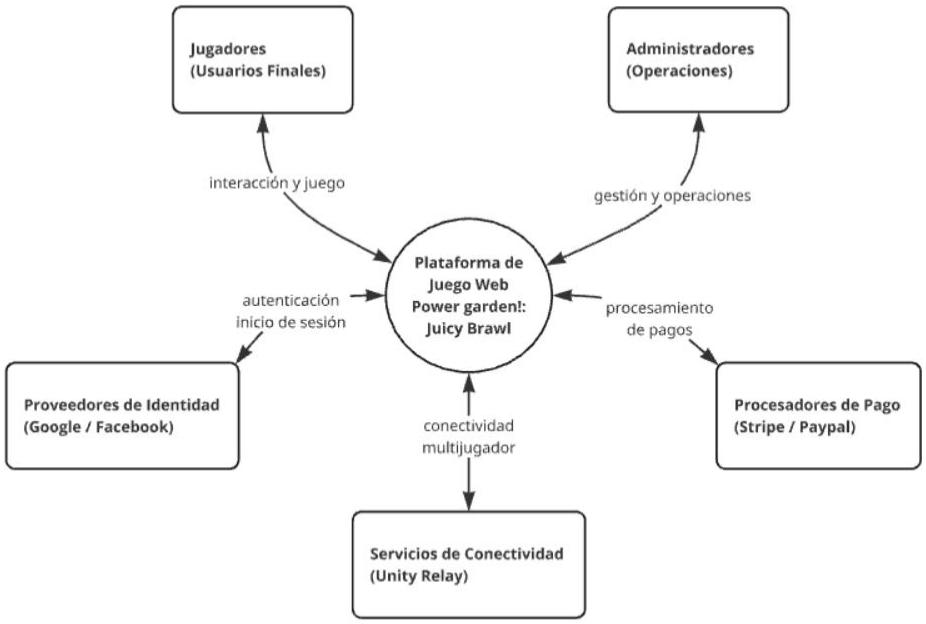
Diagrama de componentes C4 para la interfaz Web



Nota. Diagrama de componentes que representa el front-end iniciando la interacción con los demás contenedores del servicio, integrando Redis para rate-limiting, bajo un enfoque TO-BE.

Figura 7

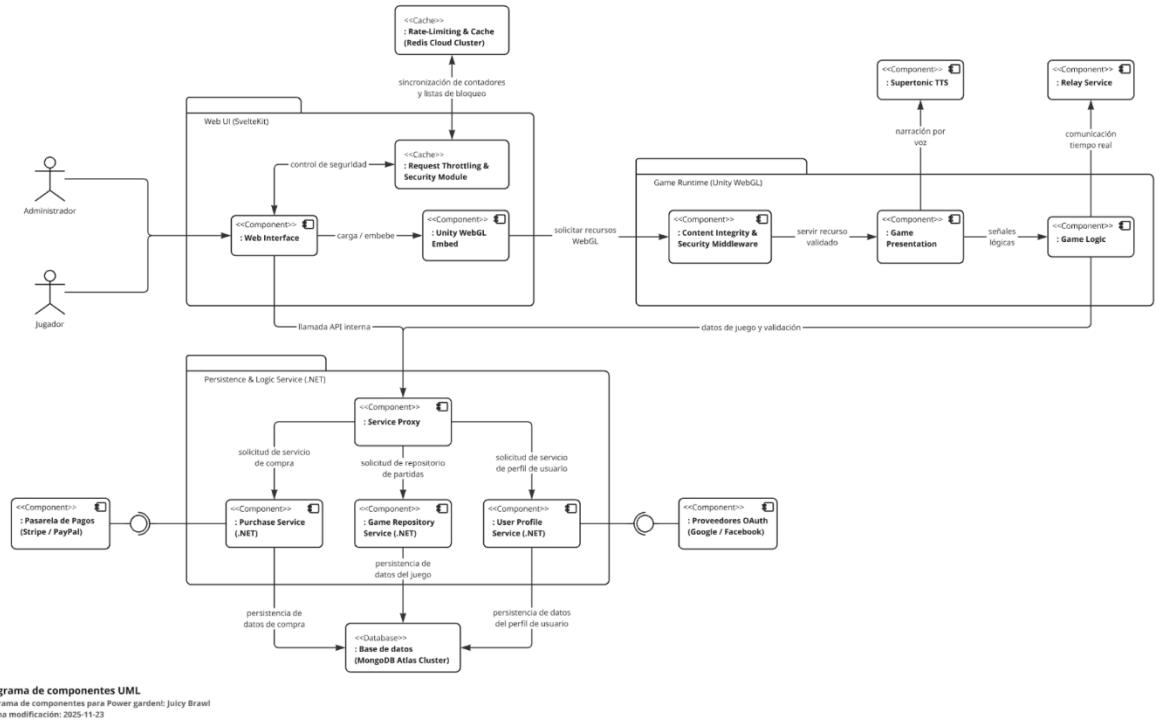
Diagrama de contexto UML



Nota. Diagrama de contexto UML de la plataforma Power Garden: Juicy Brawl, mostrando los actores externos, sistemas y la interacción general con el sistema principal.

Figura 8

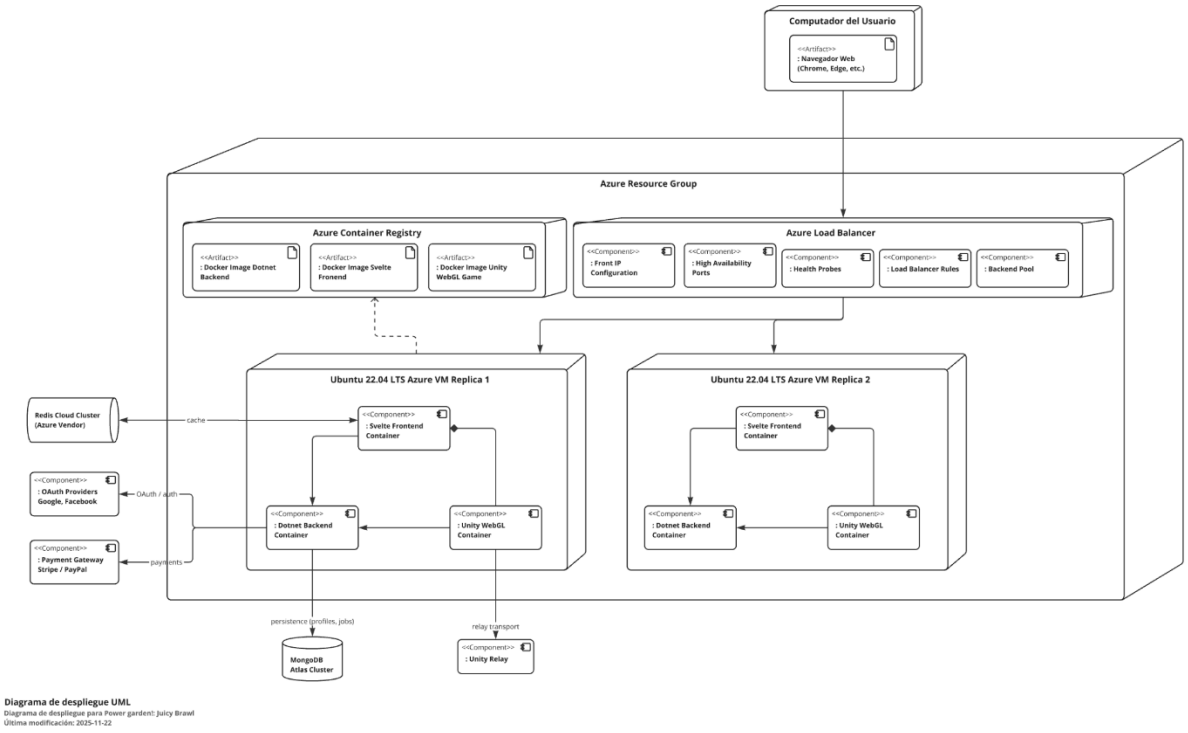
Diagrama de componentes UML



Nota. Diagrama de componentes UML que representa la interacción entre los contenedores y sus respectivos componentes, incluyendo la integración con interfaces externas.

Figura 9

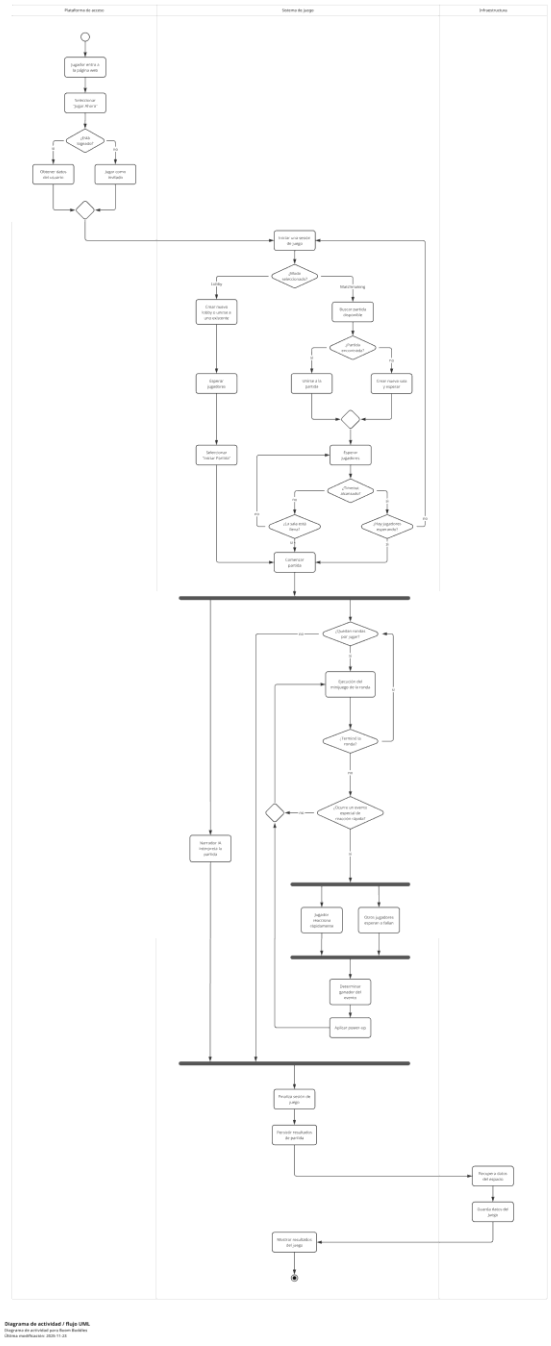
Diagrama de despliegue UML



Nota. Diagrama de despliegue que muestra un Load Balancer con health probes y VNETs configuradas, redirigiendo el tráfico a dos réplicas de VMs Ubuntu 22.04 LTS. Cada VM ejecuta tres contenedores de la aplicación completa, desplegados en Docker Swarm con imágenes obtenidas desde ACR, con dos réplicas por contenedor, y orquestados mediante Traefik como balanceador de carga interno.

Figura 10

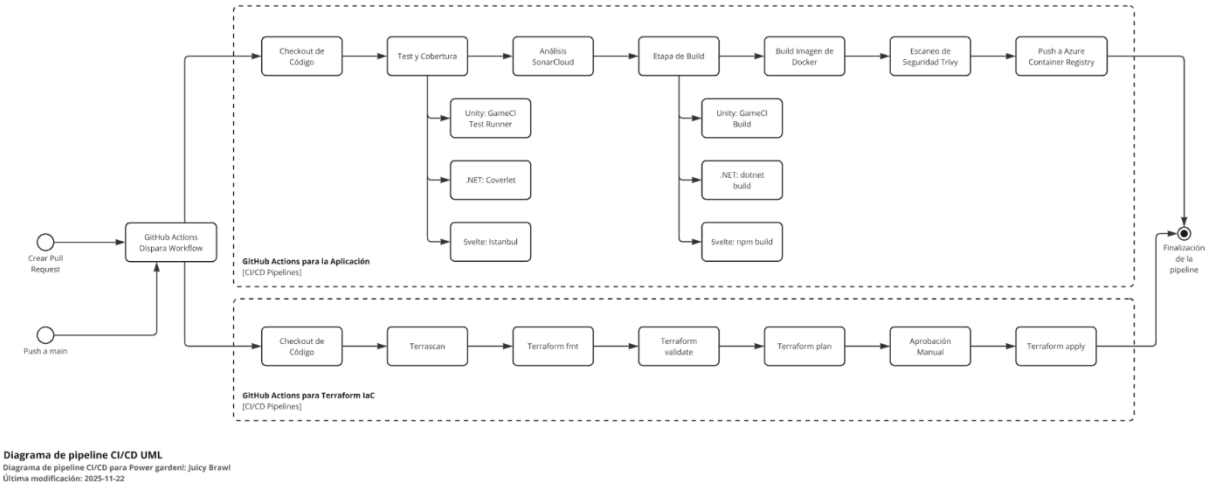
Diagrama de actividad / flujo UML



Nota. Diagrama de actividades que representa el flujo completo de la aplicación, desde que el usuario ingresa, participa en una partida y se almacena la información en la base de datos. Cada contenedor se separa mediante swimlanes para facilitar la visualización y la delegación de responsabilidades.

Figura 11

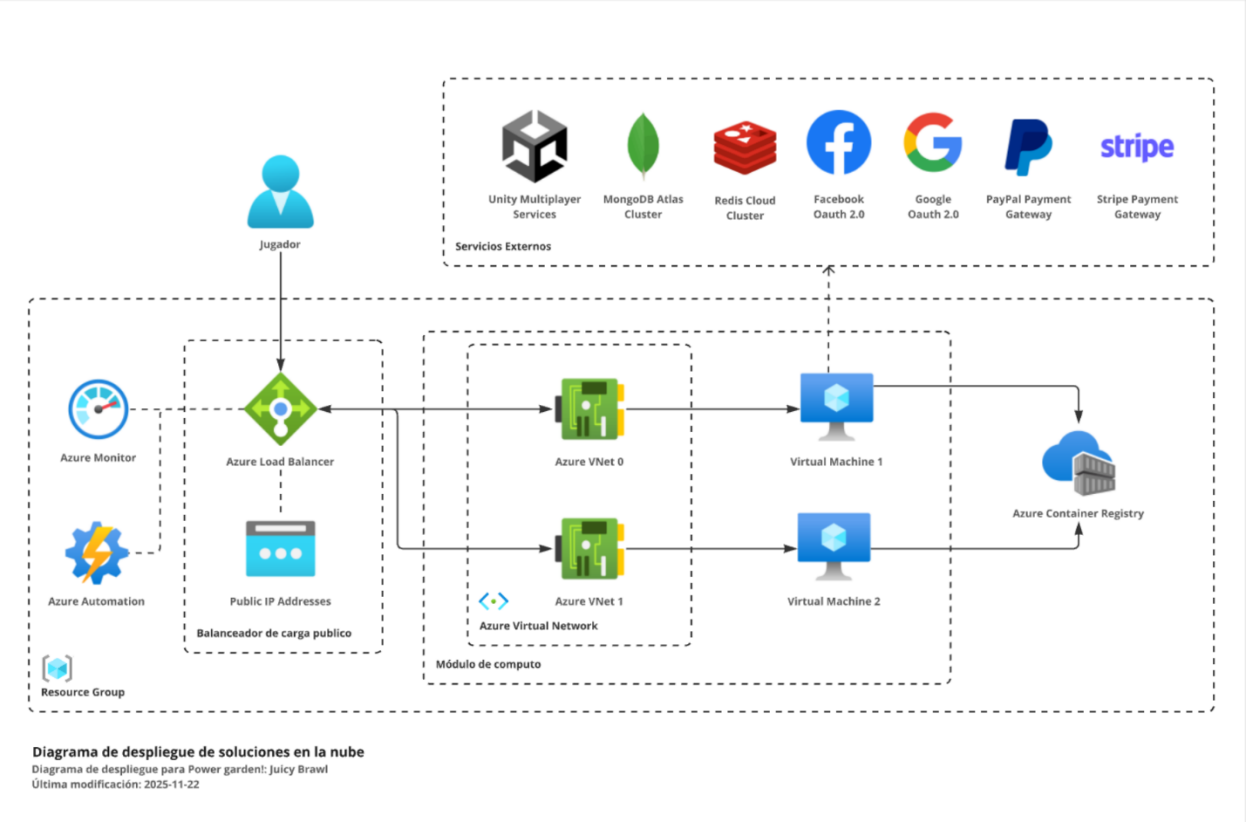
Diagrama de pipeline CI/CD UML



Nota. Diagrama que detalla los jobs a ejecutar en los pipelines de cada repositorio. Los repositorios de Back-end, Front-end y WebGL realizan checkout de código, análisis estático y dinámico, build, creación de imagen Docker, análisis de seguridad con Trivy y push a ACR. Por su parte, el repositorio de Terraform ejecuta escaneo de seguridad con Terrascan, formateo (fmt), validación, plan, aprobación manual y apply.

Figura 12

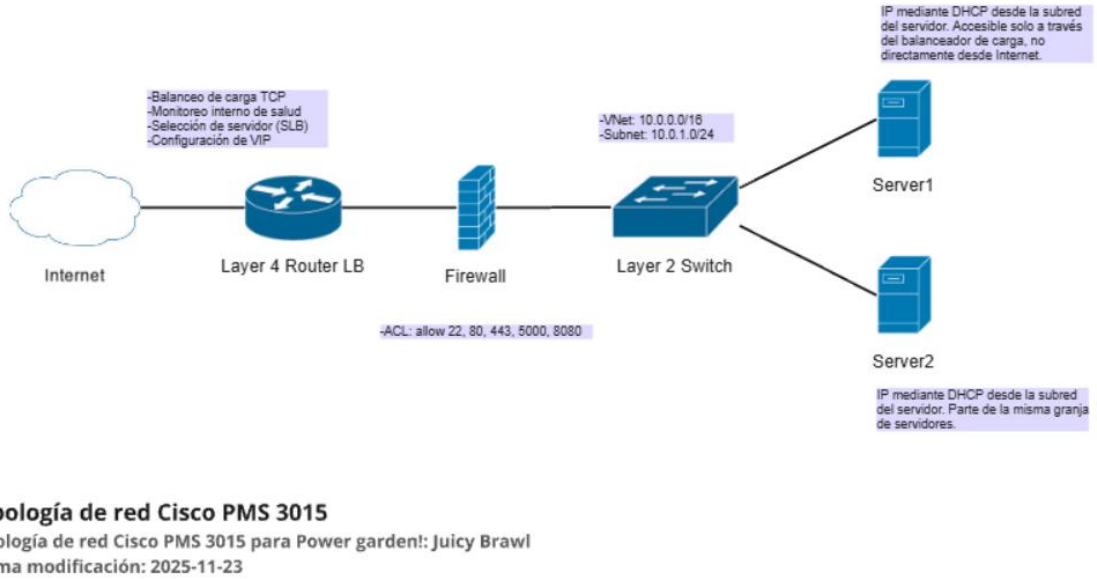
Diagrama de despliegue de soluciones en la nube



Nota. Diagrama de Azure que representa los servicios externos e internos relacionados con la aplicación, incluyendo: Azure Monitor para monitoreo de salud, Load Balancer con asignación de tres IPs públicas, Azure Automation para restauración de máquinas, Virtual Networks para el direccionamiento del LB, Azure Container Registry con las imágenes de la aplicación y la granja de cómputo con las VMs.

Figura 13

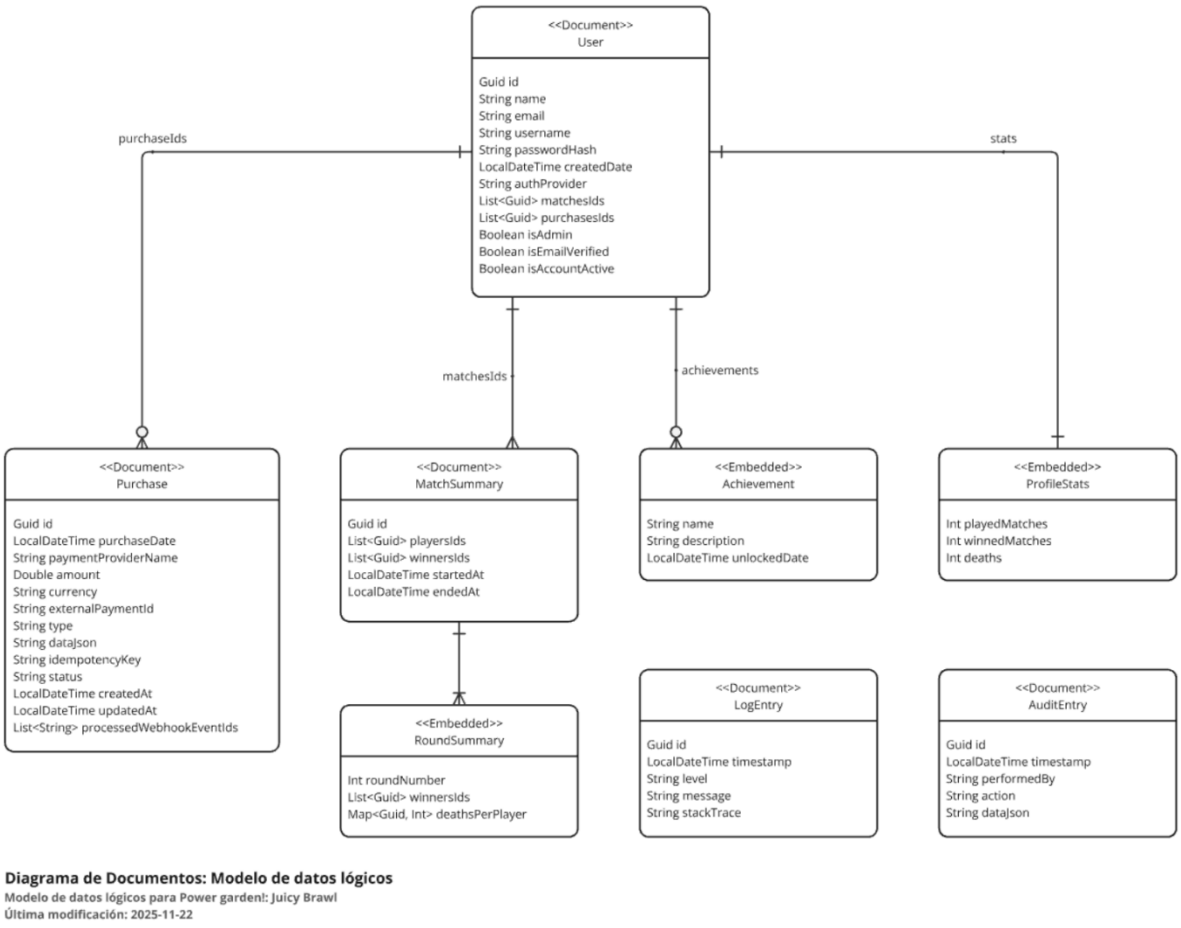
Topología de red Cisco PMS 3015



Nota. Diagrama de topología de red que evidencia el funcionamiento del Load Balancer en términos de salud y seguridad a nivel de protocolo TCP. El switch expone únicamente los puertos 22 (SSH), 80 (HTTP), 443 (HTTPS), 5000 (.NET) y 8080 (Unity). Cada VM server recibe, mediante DHCP, una IP privada dentro del rango 10.0.1.0/24 correspondiente a la subnet de la Azure VNet.

Figura 14

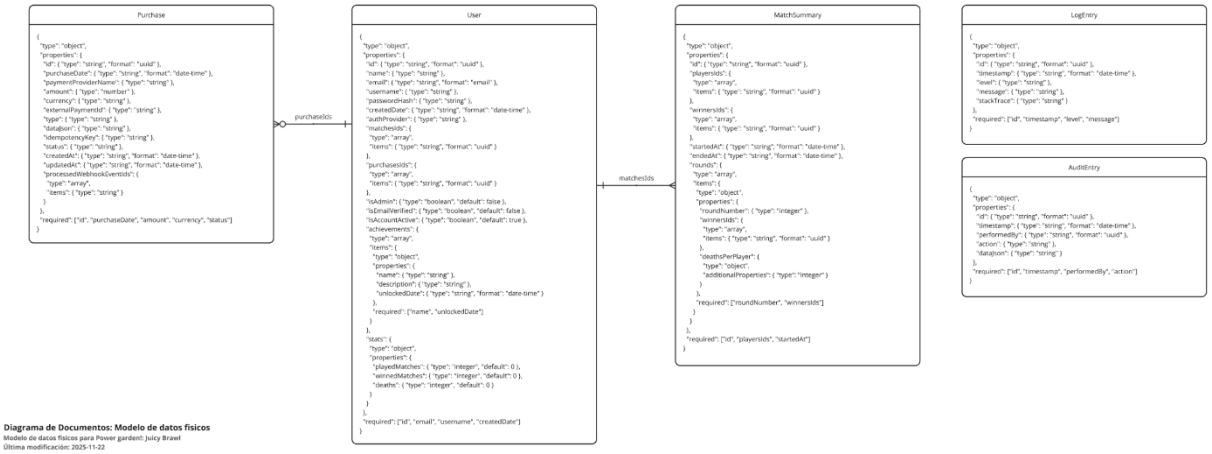
Diagrama de Documentos: Modelo de datos lógicos



Nota. Equivalente a un diagrama ER, se utiliza una base de datos no relacional MongoDB. Para ello, se siguió la guía oficial de diagramación de MongoDB para generar un diagrama de documentos que representa la capa de modelado de datos lógicos.

Figura 15

Diagrama de documentos: Modelo de datos físicos



Nota. Como paso siguiente a la documentación oficial de MongoDB, la capa lógica se transformó a la capa física, asignando los diagramas como JSON Schemas para su implementación en la base de datos.

Atributos de Calidad

Disponibilidad

Escenario de Calidad - Failover bajo carga

Fuente: 49 jugadores simultáneos.

Estímulo: Una VM o un contenedor deja de responder mientras se procesan solicitudes.

Entorno: Arquitectura distribuida con dos réplicas.

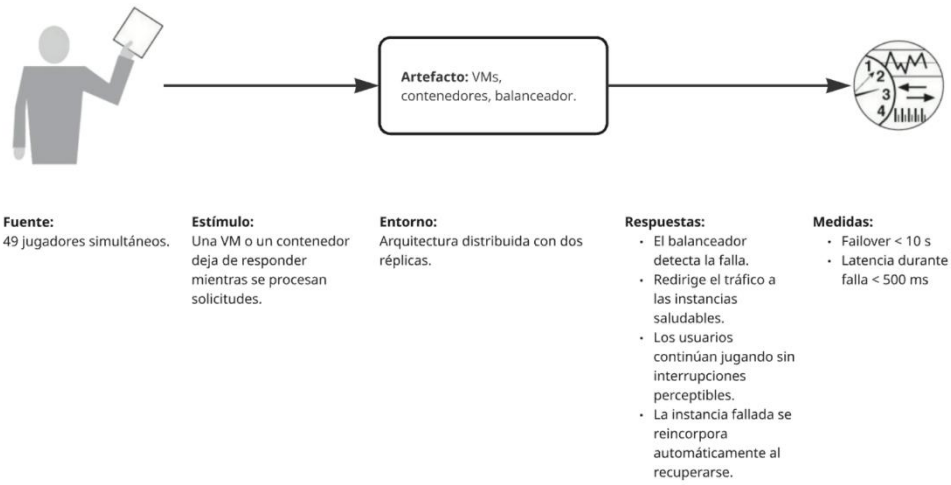
Artefactos: VMs, contenedores, balanceador.

Respuesta:

- El balanceador detecta la falla.
- Redirige el tráfico a las instancias saludables.
- Los usuarios continúan jugando sin interrupciones perceptibles.
- La instancia fallada se reincorpora automáticamente al recuperarse.

Medidas:

- Failover < 10 s
- Latencia durante la falla < 500 ms



Seguridad

Escenario de Calidad 1 - Autenticación JWT

Fuente: Jugador desde el frontend.

Estímulo: Solicita iniciar sesión.

Entorno: Frontend y backend.

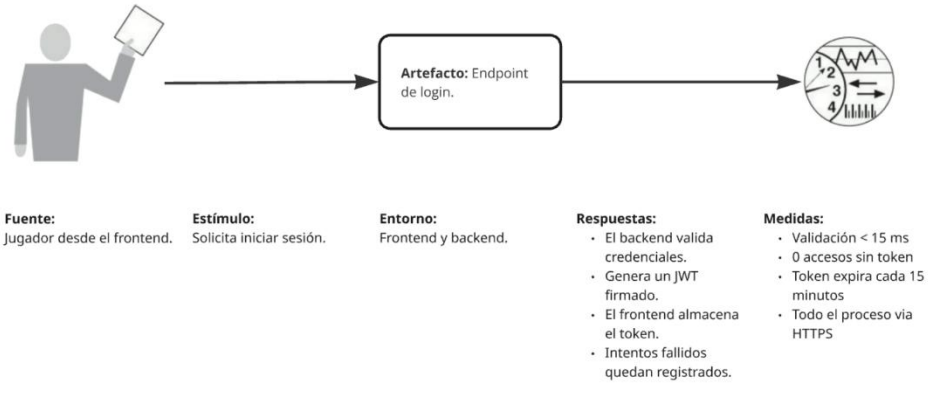
Artefacto: Endpoint de login.

Respuesta:

- El backend valida credenciales.
- Genera un JWT firmado.
- El frontend almacena el token.
- Intentos fallidos quedan registrados.

Medidas:

- Validación < 15 ms
- 0 accesos sin token
- Token expira cada 15 minutos
- Todo el proceso via HTTPS



Escenario de Calidad 2 - Autorización JWT

Fuente: Jugador ya autenticado.

Estímulo: Solicita un endpoint restringido.

Entorno: Frontend y backend.

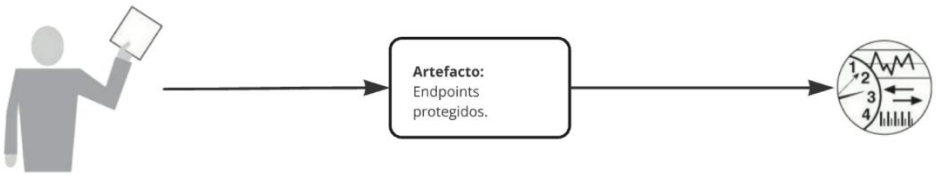
Artefacto: Endpoints protegidos.

Respuesta:

- El backend valida la firma del JWT.
- Verifica expiración y permisos.
- Permite la acción o devuelve 401/403 y registra el evento.

Medidas:

- Tiempo de autorización < 15 ms
- 100% de solicitudes inválidas bloqueadas



Fuente:
Jugador ya autenticado.

Estímulo:
Solicita un endpoint restringido.

Entorno:
Frontend y backend.

Respuestas:
- El backend valida la firma del JWT.
- Verifica expiración y permisos.
- Permite la acción o devuelve 401/403 y registra el evento.

Medidas:
- Tiempo de autorización < 15 ms
- 100% de solicitudes inválidas bloqueadas

Autorización JWT
Escenario de calidad para el atributo de "Seguridad"
Última modificación: 2025-11-28

Escenario de Calidad 3 - Integridad del Build Unity WebGL

Fuente: Atacante intentando modificar el juego.

Estímulo: Solicitud de ejecución del juego en el navegador.

Entorno: Frontend y contenedor de Unity WebGL

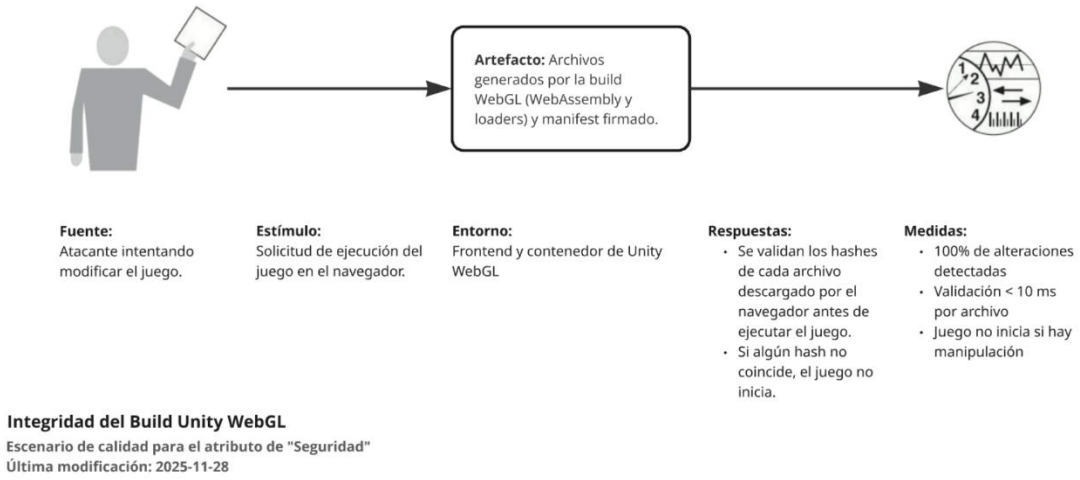
Artefacto: Archivos generados por la build WebGL (WebAssembly y loaders) y manifest firmado.

Respuesta:

- Se validan los hashes de cada archivo descargado por el navegador antes de ejecutar el juego.
- Si algún hash no coincide, el juego no inicia.

Medidas:

- 100% de alteraciones detectadas
- Validación < 10 ms por archivo
- Juego no inicia si hay manipulación



Mantenibilidad

Escenario de Calidad - Análisis Estático Continuo

Fuente: Pipeline CI.

Estímulo: Build de cualquier componente del proyecto.

Entorno: GitHub Actions.

Artefacto: Código backend, frontend y WebGL.

Respuesta:

- El análisis estático se ejecuta automáticamente.
- Si la calidad no cumple los umbrales, la PR falla.

Medidas:

- Rating A
- <5 code smells
- <3% duplicación
- 0 vulnerabilidades
- Cobertura: >40% front, >80% back

Fuente:	Estímulo:	Entorno:	Respuestas:	Medidas:
Pipeline CI.	Build de cualquier componente del proyecto.	GitHub Actions.	• El análisis estático se ejecuta automáticamente. • Si la calidad no cumple los umbrales, la PR falla.	• Rating A • <5 code smells • <3% duplicación • 0 vulnerabilidades • Cobertura: >40% front, >80% back

Análisis Estático Continuo
Escenario de calidad para el atributo de "Mantenibilidad"
Última modificación: 2025-11-28

Rendimiento

Escenario de Calidad - Latencia en Tiempo Real

Fuente: Jugadores en partida multiplayer.

Estímulo: Envían movimientos/acciones.

Entorno: Frontend de cliente, WebGL, backend del juego y servicio Unity Relay.

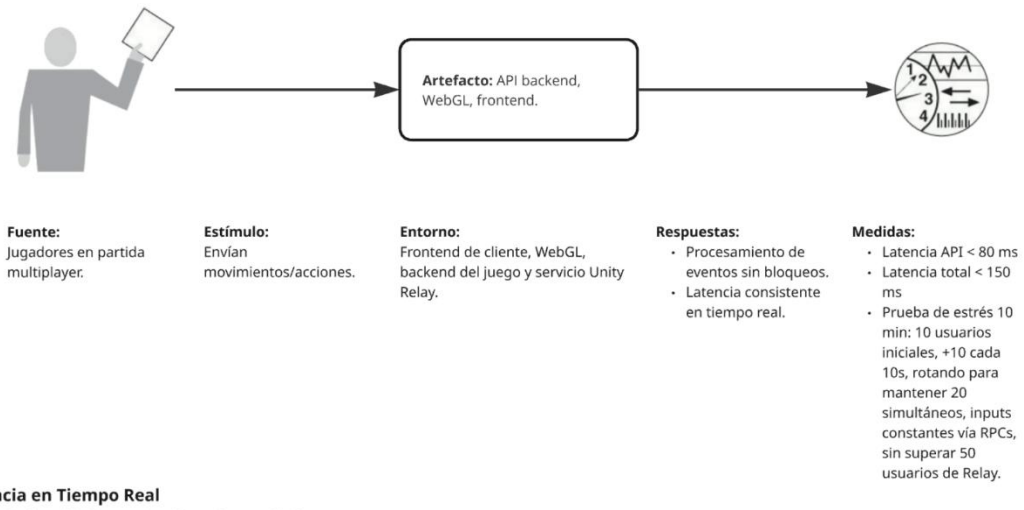
Artefactos: API backend, WebGL, frontend.

Respuesta:

- Procesamiento de eventos sin bloqueos.
- Latencia consistente en tiempo real.

Medidas:

- Latencia API < 80 ms
- Latencia total < 150 ms
- Prueba de estrés 10 min: 10 usuarios iniciales, +10 cada 10s, rotando para mantener 20 simultáneos, inputs constantes vía RPCs, sin superar 50 usuarios de Relay.



Requerimientos funcionales

1. **Concurrencia y tiempo real en el juego:**
    - **a.** El sistema debe soportar múltiples jugadores simultáneamente, garantizando interacción en tiempo real sin pérdidas de sincronización ni latencia perceptible.
    - **b.** Se deben implementar mecanismos de comunicación en tiempo real para el juego (por ejemplo, mediante WebSockets o servicios multiplayer dedicados) que permitan actualizar estados de juego de manera inmediata entre todos los participantes.

Restricciones

1. **Infraestructura:**
    a. Se debe utilizar un Load Balancer en Azure para distribuir la carga entre las instancias del backend, sin habilitar escalado automático.
    b. Las VMs, redes y contenedores deben configurarse considerando las limitaciones de Azure Free Tier donde aplique.
2. **Cronograma de desarrollo:**
    a. **Primer sprint:** duración de 2 semanas, enfocado en la implementación de la arquitectura base, la conexión inicial del juego con el backend, integración de la lógica de juego en tiempo real, persistencia de datos y pruebas de concurrencia.
    b. **Segundo sprint:** duración de 2 semanas, enfocado en el escalado horizontal del sistema y en el desarrollo de atributos de calidad como seguridad, mantenibilidad, rendimiento y disponibilidad.
3. **Licencias:**
    a. Uso del servicio Unity Relay con licencia gratuita, limitada a un máximo de 50 usuarios concurrentes jugando. Esto constituye una restricción para la cantidad de jugadores que pueden estar conectados simultáneamente al multiplayer.

Supuestos

1. **Plataforma en la nube:**
    a. Se asume que se utilizará Azure Free Tier para el desarrollo y despliegue inicial, lo que implica restricciones de recursos (CPU, memoria, número de VMs, ancho de banda).

2. **Disponibilidad de herramientas:**
    a. Se cuenta con acceso a las herramientas de desarrollo necesarias (Unity, Docker, Git, Azure DevOps o equivalente) dentro de los límites de licenciamiento y tier gratuito.
3. **Recursos del equipo:**
    a. Se asume que el equipo de desarrollo está familiarizado con las tecnologías a utilizar y puede cumplir los sprints en los tiempos establecidos.