Instituto Federal da Bahia - IFBA

Curso de Análise e Desenvolvimento de Sistemas
Campus Salvador

PhaseGen: Um Pipeline Autônomo para Geração de Assets Visuais em Jogos 2D Baseado em Modelos Generativos de Difusão

Trabalho de Conclusão de Curso

Kayran Vieira Chaves

Orientador: Prof. Antonio Carlos dos Santos Souza

Salvador, Bahia, Brasil
Julho, 2026

Sumário

1 Visão Geral

A indústria de jogos digitais vem apresentando crescimento expressivo nas últimas décadas, consolidando-se como um dos principais segmentos da economia criativa e da tecnologia. Paralelamente a esse crescimento, observa-se uma ampliação significativa da complexidade técnica e artística envolvida no desenvolvimento de jogos, impulsionada pela crescente demanda por experiências visuais mais sofisticadas, maior diversidade de conteúdo e ciclos de atualização contínuos. Nesse contexto, a produção de ativos visuais tornou-se uma das etapas mais relevantes e custosas do processo de desenvolvimento, especialmente em projetos voltados ao mercado independente.

Ao mesmo tempo, avanços recentes na área de inteligência artificial generativa vêm introduzindo novas possibilidades para automação de tarefas tradicionalmente dependentes de produção manual intensiva. Entre essas tecnologias, destacam-se os modelos generativos de imagem — incluindo arquiteturas baseadas em redes adversariais, modelos autorregressivos e modelos de difusão —, voltados à síntese visual autônoma a partir de descrições semânticas, bem como os Modelos de Linguagem de Larga Escala (Large Language Models ou LLMs), voltados à geração de conteúdo textual. A crescente maturidade dessas ferramentas ampliou sua adoção em diferentes áreas criativas, incluindo design gráfico, produção audiovisual e geração procedural de conteúdo digital, configurando um novo paradigma produtivo para áreas intensivas em criação de conteúdo.

No contexto do desenvolvimento de jogos, tais tecnologias apresentam potencial para transformar significativamente os fluxos tradicionais de produção, permitindo automatização parcial de etapas relacionadas à concepção visual, criação de personagens, geração de cenários e construção de variações artísticas. A integração entre modelos generativos e sistemas procedurais possibilita o desenvolvimento de experiências dinâmicas, nas quais o conteúdo pode ser produzido de maneira adaptativa durante o próprio ciclo de execução do software, eliminando a dependência de bibliotecas estáticas de assets e expandindo significativamente a variabilidade de conteúdo sem custo proporcional de produção manual.

Entretanto, apesar do avanço dessas tecnologias, sua aplicação prática em rotinas de produção de jogos ainda enfrenta desafios de consistência visual, integração entre ferramentas, automação de fluxos e viabilidade técnica em ambientes de desenvolvimento independente. Diante disso, o presente trabalho não propõe uma solução definitiva para a produção de assets em jogos, mas uma investigação prática de estratégias possíveis, que combina modelos de linguagem, modelos generativos de imagem e infraestrutura em nuvem em um projeto funcional concebido como vitrine tecnológica.

1.1 Declaração do Problema

O desenvolvimento de jogos digitais configura-se como uma atividade essencialmente interdisciplinar, distinguindo-se do desenvolvimento de software tradicional por demandar a integração de múltiplas áreas do conhecimento. Entre essas áreas, destacam-se a produção de artes digitais e tradicionais, composição e edição sonora, elaboração de narrativas e roteiros, level design, programação de sistemas e mecânicas, além de processos de teste e gerenciamento. Para contemplar adequadamente tais competências, torna-se necessário o envolvimento de equipes compostas por profissionais especializados em diferentes etapas do processo produtivo ou, alternativamente, de grupos reduzidos com elevada capacidade multidisciplinar. Contudo, ambos os cenários impõem desafios significativos a desenvolvedores independentes e pequenas equipes, especialmente em contextos marcados por restrições

orçamentárias e limitações de recursos humanos.

Embora a distribuição exata de força de trabalho varie conforme metodologia e porte da empresa, pesquisas como o IGDA Developer Satisfaction Survey indicam que engenharia e design concentram a maior parcela de profissionais [1], isso não diminui a centralidade da produção visual dentro do processo de desenvolvimento: mesmo compondo um número relativamente menor de profissionais, essa etapa costuma concentrar parte significativa do tempo de produção e, como discutido a seguir, é um dos aspectos mais valorizados pelo público consumidor. No mercado independente, a predominância de equipes reduzidas torna praticamente inviável sustentar uma produção artística extensa sem algum nível de automação ou terceirização.

A relevância da dimensão visual também pode ser observada sob a perspectiva do público consumidor. Em estudo conduzido por Badoni et al. (2022), verificou-se que, de forma agregada, os elementos gráficos constituem o segundo aspecto mais relevante dentro de um jogo, ficando atrás apenas da jogabilidade e à frente de áudio e mecânicas. Os autores observam ainda uma diferença de preferência associada ao gênero entre os respondentes com idade superior a 25 anos: os homens dessa faixa etária valorizam mais a jogabilidade, enquanto as mulheres valorizam mais os elementos gráficos [2]. Tais resultados indicam que a qualidade estética e gráfica de um jogo constitui diferencial competitivo importante para determinados segmentos de mercado, reforçando a necessidade de investimento em recursos artísticos durante o processo de desenvolvimento, mesmo em contextos com recursos limitados.

Diante desse quadro, a produção visual representa simultaneamente um dos maiores gargalos e um dos mais críticos fatores de qualidade no desenvolvimento independente de jogos. A busca por alternativas que reduzam esse gargalo sem comprometer a coerência estética da experiência constitui, portanto, um problema de pesquisa relevante tanto do ponto de vista técnico quanto econômico. Evidência desse cenário pode ser observada na crescente adoção de ferramentas de inteligência artificial generativa especificamente para a etapa de produção visual: a síntese qualitativa sistemática da literatura conduzida por Ternar et al. (2025), retomada em detalhe nos Trabalhos Relacionados, identificou que aproximadamente 60% das implementações de IA generativa divulgadas por jogos na plataforma Steam envolvem geração de assets visuais, indicando que os pipelines de arte constituem o principal ponto de entrada para automação na produção de jogos [3]. Embora os dados reflitam o conjunto amplo de publicadores na plataforma, a tendência é especialmente relevante para desenvolvedores independentes, para quem a automação da etapa artística representa uma das poucas alternativas viáveis diante de restrições orçamentárias e de equipe. Nesse contexto, os modelos generativos de imagem surgem como alternativa promissora, oferecendo capacidade de síntese visual autônoma a partir de descrições semânticas, com potencial de reduzir significativamente o tempo e o custo associados à produção manual de ativos gráficos.

Entretanto, a simples disponibilidade dessas tecnologias não é suficiente: a ausência de arquiteturas integradas que combinem geração textual, síntese visual e distribuição de assets em um fluxo funcional mantém esse potencial inacessível para a maior parte dos desenvolvedores independentes. É nessa lacuna que se insere a proposta do presente trabalho.

1.2 Proposta de Solução de Software

Diante do exposto, o presente projeto propõe o desenvolvimento de um pipeline automatizado de geração procedural de assets visuais, denominado PhaseGen, concebido como vitrine tecnológica para a investigação prática da integração entre Modelos de Linguagem de Larga Escala, modelos generativos de imagem e infraestrutura em nuvem no contexto do desenvolvimento de jogos digitais. O pipeline constitui o elemento central da proposta. Trata-se de um sistema autônomo capaz de atuar como um diretor de arte automatizado, produzindo personagens animados e cenários em camadas a partir de descrições semânticas geradas proceduralmente, sem intervenção humana. Dessa forma, o sistema demonstra de forma concreta como diferentes ferramentas e tecnologias podem ser combinadas para automatizar etapas tradicionalmente dependentes de produção artística manual.

Para expor os assets gerados em um contexto funcional e evidenciar sua integração com as capacidades nativas de um motor de jogo contemporâneo, o pipeline é complementado por um jogo independente do gênero *beat-em-up* 2D, desenvolvido na Godot Engine 4.5. O jogo não possui assets visuais embutidos: todos os recursos são carregados dinamicamente em tempo de execução a partir de infraestrutura em nuvem, demonstrando como a combinação entre geração procedural, armazenamento distribuído e funcionalidades do motor de jogo, como o sistema de partículas GPU, efeitos de pós-processamento por bioma e *parallax scrolling* em múltiplas camadas, pode reduzir a dependência de produção artística manual e viabilizar a criação de experiências interativas com equipes e recursos reduzidos.

O pipeline é estruturado de forma modular e opera em ciclos autônomos contínuos. Em sua etapa inicial, seleciona um bioma entre doze opções disponíveis: snow, desert, forest, jungle, swamp, cave, crystal, volcanic, graveyard, ruins, mechanical e corrupted, consultando as últimas gerações para evitar repetições consecutivas. O tipo de movimento e o tipo corporal do inimigo são determinados pelo próprio pipeline por sorteio aleatório, garantindo distribuição uniforme entre as combinações possíveis, independentemente das preferências estatísticas do modelo. Com o bioma e esses parâmetros definidos, uma chamada a um LLM produz um documento JSON estruturado que define o nome do inimigo e os prompts positivos para geração dos assets visuais, utilizando o tipo de movimento e o tipo corporal sorteados como contexto para assegurar coerência visual ao longo de toda a fase. Em paralelo, dois LLMs adicionais são acionados para geração do objetivo procedural da fase e da fala de apresentação do inimigo, ambos retornados em formato JSON estruturado.

A geração do personagem ocorre em dois jobs sequenciais — reference frame e spritesheet —, enquanto as quatro camadas de background (far, mid, intermediary e near) são produzidas em paralelo, com remoção de fundo aplicada às camadas mid e intermediary. O condicionamento por tipo corporal, o gerenciamento de memória de vídeo entre os jobs e os métodos de remoção de fundo são detalhados no Capítulo 4.

Os assets gerados são armazenados em infraestrutura de nuvem por meio do Supabase, as imagens no Supabase Storage e os metadados estruturados no banco de dados relacional PostgreSQL, e disponibilizados via Supabase Edge Functions ao motor de jogo Godot Engine 4.5. Essa abordagem possibilita a montagem dinâmica das fases durante a execução do jogo sem que qualquer asset esteja embutido no executável, desacoplando completamente o processo de geração do processo de execução.

Nesse contexto, o sistema opera de maneira análoga a um diretor de arte autônomo: reduz significativamente o intervalo entre a concepção conceitual e a implementação

funcional de uma fase jogável, sendo o jogo a materialização prática dos conceitos propostos e o meio pelo qual a viabilidade do pipeline é demonstrada em um ambiente interativo completo. As abordagens apresentadas podem ser adotadas integralmente ou aplicadas de forma parcial, permitindo que os desenvolvedores incorporem apenas as etapas mais adequadas aos objetivos, recursos disponíveis e características específicas de cada projeto.

1.3 Tecnologias Adotadas

A seleção das tecnologias empregadas no desenvolvimento do presente projeto foi orientada pela natureza interdisciplinar da proposta, priorizando soluções gratuitas ou de acesso gratuito, predominantemente open source e compatíveis com arquiteturas modulares e escaláveis. Considerando que o sistema envolve integração entre inteligência artificial generativa, geração procedural de conteúdo e execução dinâmica em tempo real, tornou-se necessário adotar ferramentas capazes de operar de forma interoperável, permitindo automação, flexibilidade de desenvolvimento e expansão futura do sistema.

1.3.1 Linguagem e Orquestração do Pipeline

Python 3.10

A linguagem Python foi adotada como principal tecnologia para implementação do pipeline de geração procedural [4]. Sua escolha fundamenta-se na ampla utilização em aplicações de inteligência artificial, automação e integração entre serviços distribuídos, além de um ecossistema consolidado de bibliotecas para comunicação com APIs, manipulação de estruturas JSON e processamento de dados.

No projeto, Python atua como elemento de orquestração central, responsável pela comunicação entre os modelos de linguagem, o ambiente de geração visual ComfyUI, a infraestrutura de persistência Supabase e o servidor de dashboard. O servidor principal é implementado com o framework FastAPI e executado via Uvicorn como servidor ASGI, expondo endpoints REST para controle do pipeline e um endpoint WebSocket para transmissão de eventos em tempo real ao dashboard. A execução do ciclo de geração adota um modelo assíncrono que preserva a responsividade do servidor durante operações de longa duração, cujo funcionamento é detalhado no Capítulo 4.

A versão 3.10 foi adotada especificamente em razão de sua compatibilidade consolidada com as principais dependências do stack de inteligência artificial utilizado no projeto incluindo PyTorch e o ambiente ComfyUI, cujo suporte a versões mais recentes do interpretador apresentava instabilidades no período de desenvolvimento.

O dashboard de monitoramento é implementado como uma página HTML estática servida diretamente pelo servidor FastAPI, comunicando-se com o backend via WebSocket para recebimento de eventos em tempo real sem dependência de frameworks frontend adicionais.

1.3.2 Modelos de Linguagem

Llama 3.1 8B Instant (via Groq), Gemini 2.0 Flash (via Google) e OpenRouter

No contexto da geração procedural de conteúdo textual, o projeto utiliza três provedores de modelos de linguagem em arquitetura de rotação automática: o modelo Llama 3.1 8B Instant, acessado via API da Groq [5]; o Gemini 2.0 Flash, acessado via API da Google [6]; e modelos gratuitos disponibilizados via OpenRouter, como DeepSeek

e Gemma [7]. A adoção de múltiplos provedores justifica-se pela necessidade de garantir continuidade operacional do pipeline: cada provedor possui limites de requisições distintos e pode estar temporariamente indisponível, de modo que a rotação automática entre eles elimina a dependência de um único serviço externo.

A utilização de LLMs justifica-se pela necessidade de produzir descrições semânticas dinâmicas capazes de definir temáticas, estilos visuais e parâmetros estruturais das fases do jogo. Diferentemente de abordagens baseadas em regras fixas ou bancos estáticos de conteúdo, os modelos de linguagem permitem geração contextualizada e procedural, ampliando significativamente a variabilidade e a escalabilidade do sistema. Cada chamada ao LLM produz um documento JSON estruturado contendo os dados do inimigo e os prompts de geração visual, garantindo coerência temática entre todos os ativos da fase.

Entre os três, o Llama 3.1 8B Instant destaca-se pelo equilíbrio entre qualidade semântica e velocidade de inferência, enquanto o Gemini 2.0 Flash e o OpenRouter contribuem principalmente para ampliar a cota agregada de requisições disponíveis ao pipeline, reduzindo a probabilidade de todos os provedores estarem simultaneamente indisponíveis. O mecanismo de rotação e cooldown que coordena os três provedores é detalhado na Visão Arquitetural do Capítulo 4.

1.3.3 Modelos de Difusão

Stable Diffusion 1.5

Para a geração dos ativos visuais, o projeto utiliza o modelo Stable Diffusion 1.5, baseado na arquitetura de difusão latente proposta por Rombach et al. (2022) [8]. A escolha desse modelo fundamenta-se no equilíbrio entre qualidade visual, flexibilidade de customização e viabilidade computacional. Em comparação com modelos mais recentes, que frequentemente demandam maior capacidade de processamento gráfico e memória de vídeo, o Stable Diffusion 1.5 apresenta desempenho adequado mesmo em infraestruturas de hardware mais acessíveis, característica particularmente relevante no contexto de desenvolvimento independente, onde a disponibilidade de recursos computacionais de alto desempenho não pode ser assumida. A Tabela 1 ilustra essa diferença, comparando os requisitos de VRAM entre os principais modelos de difusão disponíveis atualmente. O aumento de requisito no SDXL decorre de sua arquitetura substancialmente maior — um backbone UNet cerca de três vezes maior e um segundo codificador de texto —, além do modelo de refinamento aplicado em cascata [11], o que eleva o consumo de memória em relação ao Stable Diffusion 1.5 e reforça a adequação deste último a hardware mais modesto. É importante ressaltar que essa escolha fundamenta-se em requisitos documentados de VRAM e não em experimento comparativo de qualidade visual entre os modelos listados na Tabela 1; uma comparação empírica sistemática entre eles permanece como direção de pesquisa futura, discutida na Seção 8.1.

Tabela 1: Comparação de requisitos de VRAM entre modelos de difusão [9, 10, 11]

Modelo	**VRAM Recomendada**	**Observações**
Stable Diffusion 1.5	8 GB	Melhor desempenho em hardware mais antigo e acessível.
Stable Diffusion XL 1.0	16 GB	Resolução superior (1024×1024); exige mais memória para o Refiner e LoRAs.
Stable Diffusion 3.5 (Large)	24 GB (FP16)	Propenso a erros de memória em GPUs de baixo porte sem configurações específicas.
Flux.1 (Schnell)	16 GB+	Utiliza encoder de texto T5 de grande porte; altamente detalhado, mas intensivo em memória.
Flux.1 (Dev)	24 GB+	Qualidade superior entre os modelos listados; idealmente requer GPUs A100 ou RTX 4090.

Outro fator determinante para sua adoção consiste em sua ampla consolidação dentro da comunidade open source, o que proporciona acesso a extensa quantidade de modelos derivados, extensões e técnicas de ajuste fino (fine-tuning), recursos que viabilizam a especialização do modelo base para domínios visuais específicos sem necessidade de treinamento completo.

No projeto, o Stable Diffusion 1.5 é utilizado em conjunto com componentes complementares que orientam e refinam o processo de geração, apresentados na Tabela 2. A forma como esses componentes são integrados aos workflows de geração é descrita no Capítulo 4.

Tabela 2: Componentes complementares ao Stable Diffusion 1.5

Tipo	**Componente**	**Função**
Checkpoint	ReV Animated [12]	Especializado na geração de personagens humanoides
LoRA	aziibpixelmix [13]	Orienta as gerações para o estilo de *pixel art* 16-bit
LoRA	aziib_pixel_style [14]	Reforça a coerência estilística com a estética do jogo
VAE	OrangeMixs [15]	Produz cores mais vibrantes e contornos mais definidos em relação ao VAE padrão

A combinação desses quatro componentes sobre o modelo base pode ser observada diretamente no resultado da geração, exemplificado na Figura 1.

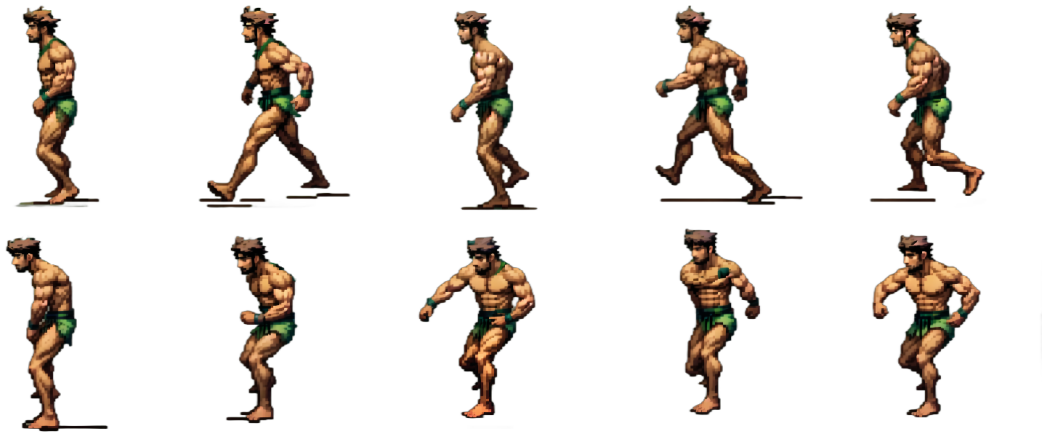

Figura 1: Exemplo de *spritesheet* gerado autonomamente pelo pipeline, produzido com o checkpoint ReV Animated, os adaptadores LoRA aziibpixelmix e aziib_pixel_style, e o VAE OrangeMixs sobre o Stable Diffusion 1.5.

1.3.4 Ambiente de Orquestração Visual

ComfyUI 0.3.66

O ComfyUI foi adotado como ambiente responsável pela orquestração dos fluxos de geração de imagem [16]. Sua arquitetura baseada em grafos de nós (node-based workflows) permite a construção modular de pipelines complexos de inferência, com controle granular sobre cada etapa do processo de geração visual. Essa característica torna o ComfyUI particularmente adequado para o presente projeto, no qual diferentes combinações de modelos, referências visuais e parâmetros precisam ser aplicadas de forma automatizada e reprodutível a cada ciclo de geração.

No projeto, o ComfyUI é controlado inteiramente via sua API REST local, mantendo seis workflows distintos: um para a imagem de referência do inimigo (reference frame), um para a folha de sprites de animação (spritesheet) e quatro para as camadas de background: far, mid, intermediary e near. Os detalhes da comunicação com o servidor — carregamento e parametrização dinâmica dos workflows, submissão dos jobs e acompanhamento da execução — são apresentados na Visão Arquitetural do Capítulo 4.

Para a remoção de fundo dos assets de personagem, o pipeline utiliza o custom node ComfyUI-BRIA_AI-RMBG [17], baseado no modelo BRIA RMBG, especializado em segmentação semântica de primeiro plano. As razões para adotar a segmentação semântica em vez do *chroma key* nesses assets são discutidas nas Decisões Arquiteturais do Capítulo 4.

O Apêndice A apresenta o workflow de geração do *spritesheet* conforme configurado no ComfyUI.

1.3.5 Mecanismos de Controle de Consistência

ControlNet e IP-Adapter

Uma das principais limitações dos modelos de difusão tradicionais consiste na dificuldade de manter consistência estrutural e estilística entre gerações consecutivas. Em aplicações voltadas à produção de ativos para jogos digitais, essa limitação pode resultar em variações excessivas entre quadros de animação ou entre o personagem e o cenário, comprometendo a coerência visual da experiência. Com o objetivo de mitigar esse problema, o projeto utiliza os mecanismos de condicionamento ControlNet, proposto

por Zhang, Rao e Agrawala (2023) [18], e IP-Adapter [19], integrados aos workflows do ComfyUI.

O ControlNet é uma extensão arquitetural dos modelos de difusão que introduz condicionamento estrutural adicional ao processo de geração, permitindo que mapas de controle, como mapas de profundidade, mapas de pose e bordas detectadas, guiem a composição espacial da imagem gerada (Figura 2). Os dois mapas empregados no pipeline capturam aspectos distintos e complementares da estrutura da imagem: o mapa de profundidade estima a distância relativa de cada região da cena, definindo o volume e a disposição espacial dos elementos; já o mapa de pose OpenPose [21] detecta a posição de articulações e membros a partir de uma imagem de referência, impondo uma postura corporal específica à figura gerada. Essa distinção determina o uso de cada mapa dentro do pipeline: na geração do personagem, profundidade e pose são aplicados em conjunto como imagens de controle na geração do *reference frame* — a profundidade conferindo volume e estrutura corporal, e o OpenPose impondo a postura — e essa referência é utilizada como condição adicional de entrada na geração do *spritesheet*, garantindo que as poses de animação preservem a identidade visual do personagem recém-criado; já na geração das camadas de background, apenas o mapa de profundidade é utilizado, por corresponder diretamente à noção de profundidade perceptual que estrutura o efeito de *parallax* entre as quatro camadas do jogo. Essa abordagem, onde o condicionamento estrutural é adaptado ao tipo de asset sendo gerado, é central para a consistência visual do conjunto de assets produzidos em um mesmo ciclo.

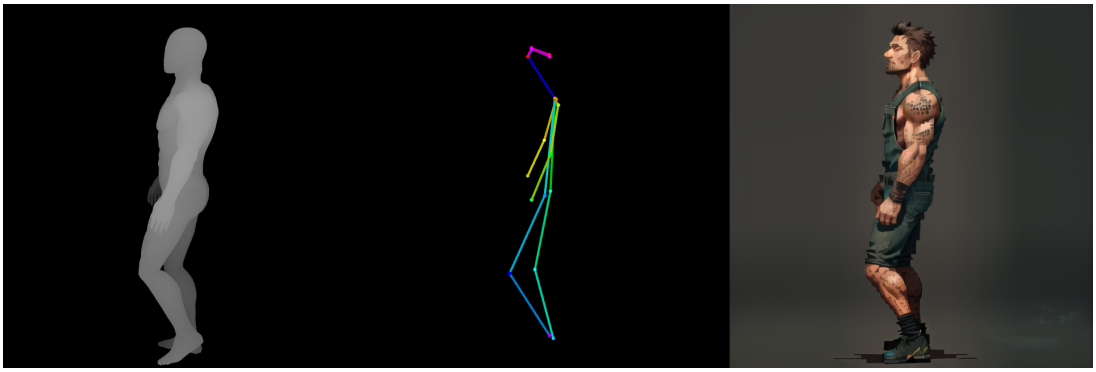

Figura 2: Imagens de controle utilizadas pelo ControlNet na geração do personagem: mapa de profundidade (depth), mapa de pose (OpenPose) e resultado final gerado pelo pipeline.

O IP-Adapter é um adaptador que opera no espaço de embeddings visuais, permitindo transferir características de aparência e estilo de uma imagem de referência para o processo de geração sem modificar os pesos do modelo base. Atua como mecanismo complementar ao ControlNet: enquanto este controla a estrutura da imagem gerada, pose, geometria e profundidade, o IP-Adapter preserva a identidade visual do personagem entre os dois assets gerados sequencialmente, utilizando o *reference frame* como referência durante a geração do *spritesheet* para manter consistência de paleta de cores, textura e estilo artístico, como ilustrado na Figura 3.

Figura 3: Comparação entre a imagem de referência (*reference frame*) e o *spritesheet* de animação gerados sequencialmente pelo pipeline para o mesmo inimigo, evidenciando a consistência visual obtida pelo condicionamento via ControlNet e IP-Adapter.

Supabase

No que se refere à infraestrutura de persistência e distribuição de dados, o projeto utiliza o Supabase como solução de backend [20]. A plataforma oferece uma arquitetura integrada baseada em PostgreSQL, armazenamento de objetos em nuvem (Supabase Storage) e funções serverless executadas sob demanda (Supabase Edge Functions), constituindo uma solução completa para persistência, armazenamento e distribuição dos ativos gerados pelo pipeline.

A adoção do Supabase fundamenta-se em três características complementares. Primeiro, sua natureza open source e o plano gratuito disponível tornam a plataforma viável para projetos independentes sem infraestrutura de backend dedicada. Segundo, o uso de PostgreSQL como banco de dados relacional nativo oferece suporte a tipos avançados como JSONB, utilizado para o campo de objetivo procedural, e restrições CHECK, que constituem uma camada adicional de validação de dados independente do código da aplicação. Terceiro, a arquitetura da plataforma permite separar completamente as credenciais de escrita, restritas ao backend de geração, das interfaces de leitura expostas ao cliente, garantindo que nenhuma credencial privilegiada seja transmitida ao motor de jogo.

Os ativos gerados são persistidos em duas camadas distintas: as imagens PNG com canal alfa para as camadas que passaram pelo processo de remoção de fundo são armazenadas no Supabase Storage, retornando URLs públicas para acesso direto pelo cliente; os metadados estruturados, incluindo os prompts utilizados, o provedor LLM responsável, o tipo corporal e de movimento do inimigo, o objetivo procedural da fase e a fala de introdução do inimigo são persistidos nas tabelas relacionais do PostgreSQL, organizadas em cinco tabelas normalizadas: phases, enemy_types, sprites, phase_enemies e background_layers.

Supabase Edge Functions

As Supabase Edge Functions constituem a interface exclusiva de leitura exposta ao cliente de jogo. A função /functions/v1/phases retorna a lista resumida de todas as fases disponíveis, enquanto a função /functions/v1/phase agrega, em uma única requisição, os dados de múltiplas tabelas em um documento JSON composto, reduzindo a latência de carregamento e eliminando a necessidade de o cliente acessar o banco de dados diretamente. Essa abordagem oferece uma camada adicional de segurança, pois o cliente nunca interage com a API PostgREST diretamente, ficando exposto apenas às funções serverless com escopo de dados controlado.

1.3.6 Motor Gráfico

Godot Engine 4.5

Como motor gráfico, foi utilizada a Godot Engine 4.5 [22]. Sua escolha fundamenta-se em seu caráter open source sob licença MIT, ausência de custos de licenciamento e elevada flexibilidade para integração com serviços externos via HTTP, característica essencial para o consumo dinâmico dos ativos persistidos em nuvem. Estudo conduzido por Holfeld (2024) sobre a relevância do Godot no mercado independente identificou crescimento expressivo de sua adoção na plataforma itch.io, predominantemente indie, passando de 2,5% em 2018 para 7,51% em 2023, posicionando-o como o terceiro motor mais utilizado no segmento [23]. Essa trajetória reflete tanto a maturidade técnica da engine quanto sua adequação ao perfil de desenvolvimento independente, contexto diretamente alinhado ao presente projeto. A engine oferece suporte nativo ao desenvolvimento de jogos 2D, sistema de cenas hierárquico, linguagem de script própria (GDScript) e recursos nativos relevantes para o projeto, como sistema de partículas GPU, efeitos de pós-processamento e suporte a *spritesheet*s dinâmicos.

No projeto, o cliente de jogo é desenvolvido inteiramente em GDScript, sem nenhum ativo visual embutido no executável: todos os recursos são carregados dinamicamente em tempo de execução a partir da infraestrutura Supabase. Os detalhes de implementação do cliente, incluindo o sistema de *parallax scrolling*, os efeitos visuais por bioma e a arquitetura de singletons globais estão descritos no Capítulo 4. A Figura 4 apresenta uma fase gerada pelo pipeline em execução no motor de jogo.

Figura 4: Fase em execução no motor de jogo Godot Engine 4.5, com os assets gerados pelo pipeline carregados dinamicamente em tempo de execução.

1.3.7 Controle de Versão e Gerenciamento de Código

GitHub

Para o controle de versão e gerenciamento do código-fonte do projeto, foi utilizada a plataforma GitHub [24], adotada por sua ampla integração com ferramentas de desenvolvimento, visibilidade pública do repositório e compatibilidade com fluxos de

trabalho colaborativos. O repositório organiza o projeto em três módulos principais: backend/, contendo o pipeline de geração em Python; phasegen-godot/, contendo o cliente de jogo; e supabase/, contendo as Edge Functions. Essa organização modular favorece a rastreabilidade das alterações, o isolamento de responsabilidades entre os subsistemas e a manutenibilidade do projeto ao longo do ciclo de desenvolvimento.

1.3.8 Modelagem e Documentação Técnica

PlantUML

O PlantUML foi adotado como ferramenta para elaboração dos diagramas UML que compõem a documentação técnica do projeto [25]. Sua escolha fundamenta-se na abordagem de modelagem orientada a texto: os diagramas são descritos por meio de uma linguagem de marcação própria e renderizados automaticamente a partir dessas descrições, eliminando a necessidade de posicionamento manual de elementos e favorecendo a rastreabilidade das alterações ao longo do desenvolvimento.

No projeto, o PlantUML foi utilizado localmente para a geração de cinco categorias de diagramas: o diagrama de caso de uso, que delimita os atores e as interações de alto nível com o sistema; os diagramas de classes do backend Python e do cliente Godot, que descrevem a estrutura estática dos subsistemas; os diagramas de atividade, que documentam o fluxo de controle do pipeline de geração e seus mecanismos de resiliência; o diagrama de componentes, que ilustra a organização arquitetural do sistema e os protocolos de comunicação entre os subsistemas; e o diagrama entidade-relacionamento, que representa o modelo relacional do banco de dados. Todos os diagramas produzidos estão apresentados no Capítulo 4.

1.4 Trabalhos Relacionados

O avanço recente das técnicas de inteligência artificial generativa tem impulsionado novas abordagens para geração procedural de conteúdo em jogos digitais. Em especial, a integração entre modelos de linguagem de larga escala e modelos generativos visuais vem ampliando as possibilidades de automação de processos tradicionalmente dependentes de produção manual intensiva. Nesse contexto, diferentes pesquisas passaram a investigar o uso dessas tecnologias tanto para geração de narrativas e mecânicas quanto para produção de ativos gráficos e estruturas completas de jogos.

1.4.1 Procedural Content Generation via Generative Artificial Intelligence (2024)

O trabalho de Mao et al. (2024) [26] investiga o impacto dos modelos generativos modernos no contexto de geração procedural de conteúdo, analisando como arquiteturas contemporâneas, incluindo modelos de linguagem e modelos de difusão, vêm sendo incorporadas a sistemas de PCG para automatizar processos criativos no desenvolvimento de jogos. Os autores destacam que essas abordagens permitem maior flexibilidade semântica na geração de conteúdo em relação a técnicas procedurais tradicionais baseadas em regras determinísticas, e enfatizam seu potencial para a democratização do desenvolvimento independente em cenários com recursos limitados. Entretanto, o estudo concentra-se na análise das tecnologias de forma isolada, sem propor ou demonstrar uma arquitetura integrada que as combine em um pipeline funcional completo. O PhaseGen parte exatamente dessa

lacuna, articulando LLMs e modelos de difusão em um sistema operacional unificado voltado à produção autônoma de assets para jogos.

1.4.2 Procedural Content Generation in Games: A Survey with Insights on Emerging LLM Integration (2024)

O estudo de Farrokhi Maleki e Zhao (2024) [27] apresenta uma revisão sistemática sobre a evolução das técnicas de PCG com foco específico na incorporação de LLMs em pipelines de geração procedural. Os autores argumentam que a integração entre LLMs e sistemas procedurais representa uma mudança significativa no paradigma de geração de conteúdo, permitindo maior coerência temática e adaptabilidade durante a criação automática de experiências interativas. O trabalho, contudo, restringe-se ao domínio textual e semântico da geração procedural, não abordando a integração com modelos generativos visuais nem a distribuição dos conteúdos produzidos a um ambiente de execução em tempo real. O PhaseGen estende essa perspectiva ao utilizar LLMs não apenas como geradores de descrições, mas como coordenadores semânticos de um pipeline que inclui síntese visual, pós-processamento de imagens e distribuição de assets via infraestrutura em nuvem.

1.4.3 Intelligent Generation of Graphical Game Assets (2023)

No contexto específico da geração de ativos gráficos, Fukaya et al. (2023) [28] realizam uma análise abrangente das técnicas utilizadas para criação automatizada de elementos visuais em jogos digitais, explorando abordagens baseadas em aprendizado profundo e modelos generativos aplicadas a personagens, cenários e componentes estéticos. Os autores identificam como desafios centrais a consistência visual entre assets, o controle estilístico e a integração prática dos elementos produzidos em pipelines reais de desenvolvimento sem, no entanto, propor soluções concretas para esses problemas em um ambiente funcional. O PhaseGen endereça diretamente esses desafios por meio dos mecanismos de condicionamento ControlNet e IP-Adapter, do processo de remoção de fundo adaptado por tipo de asset e da integração dos resultados a um motor de jogo em tempo de execução.

1.4.4 Generative AI in Game Development: A Qualitative Research Synthesis (2025)

Ternar et al. (2025) [3] conduzem uma síntese qualitativa sistemática da literatura sobre adoção de IA generativa no desenvolvimento de jogos, cobrindo estudos publicados entre 2020 e 2025. Conforme já discutido na Declaração do Problema, o trabalho identifica os pipelines de arte como o principal ponto de entrada para automação na produção de jogos. Os autores observam, contudo, que a maior parte dos estudos analisados investiga contextos isolados, game jams, protótipos de curta duração ou estúdios individuais com escopo restrito e sem demonstração de viabilidade em pipelines completos de produção. O PhaseGen situa-se precisamente nessa lacuna identificada pelos autores, propondo e implementando um pipeline end-to-end operacional que cobre desde a geração textual até a disponibilização dos assets em um ambiente interativo funcional.

1.4.5 Correlação Entre os Trabalhos e Diferenciais do PhaseGen

Os trabalhos analisados convergem em dois pontos centrais: a crescente adoção de IA generativa no desenvolvimento de jogos e o reconhecimento de LLMs como mecanismos

eficazes de coordenação semântica em pipelines procedurais. Entretanto, observa-se uma lacuna consistente entre os estudos: a maior parte concentra-se em análises conceituais, revisões sistemáticas ou experimentos de escopo restrito, sem demonstração de viabilidade em ambientes funcionais completos que integrem geração textual, síntese visual e execução em tempo real.

A Tabela 3 resume essas diferenças de escopo entre os trabalhos analisados e o PhaseGen.

Tabela 3: Comparação entre trabalhos relacionados e o PhaseGen

Autor	**Ano**	**IA utilizada**	**Ger. textual**	**Ger. visual**	**Pipeline completo**	**Exec. tempo real**	**Diferencial**
Mao et al.	2024	LLM + Difusão	Não	Não	Não	Não	Análise conceitual isolada
Farrokhi Maleki & Zhao	2024	LLM	Sim	Não	Não	Não	Revisão focada em texto
Fukaya et al.	2023	Difusão/DL	Não	Sim	Não	Não	Revisão de técnicas visuais
Ternar et al.	2025	Diversas	—	—	—	—	Síntese qualitativa da adoção
PhaseGen	2026	LLM + Difusão	Sim	Sim	Sim	Sim	Pipeline *end-to-end* funcional

O PhaseGen diferencia-se desses trabalhos ao propor e implementar um pipeline completo e funcional voltado especificamente à geração procedural de ativos visuais 2D em estilo pixel art, integrados dinamicamente a uma game engine em tempo de execução. A proposta combina modelos de linguagem com rotação automática entre provedores, modelos de difusão controlados por workflows parametrizados, mecanismos de condicionamento visual (ControlNet e IP-Adapter), processamento de imagem com métodos de remoção de fundo adaptados por tipo de asset e infraestrutura distribuída em nuvem em uma arquitetura operacional unificada. Diferentemente dos trabalhos correlatos, o sistema não apenas explora conceitos de geração procedural assistida por inteligência artificial, mas demonstra sua viabilidade em um ambiente funcional end-to-end, da descrição textual à fase jogável, sem intervenção humana após a inicialização do pipeline. Com isso, o presente trabalho responde diretamente à lacuna identificada por Ternar et al. (2025) e complementa as perspectivas teóricas de Mao et al. (2024) e Farrokhi Maleki e Zhao (2024) com uma implementação concreta e verificável. É a partir dessas lacunas identificadas na literatura que os objetivos do presente trabalho são definidos, conforme apresentado na seção seguinte.

1.5 Objetivos

1.5.1 Objetivo Geral

O presente trabalho tem como objetivo geral investigar a viabilidade de integrar modelos de linguagem de larga escala e modelos generativos de imagem em um pipeline automatizado de geração procedural de assets visuais para jogos digitais, materializada no desenvolvimento de um sistema funcional denominado PhaseGen. O pipeline articula LLMs e modelos de difusão para produção autônoma de personagens, animações e cenários, sem intervenção humana após sua inicialização, constituindo uma das estratégias possíveis para incorporar inteligência artificial generativa à rotina de desenvolvimento independente de jogos. Um jogo funcional do gênero *beat-em-up* 2D, desenvolvido na Godot Engine 4.5, complementa a proposta ao expor os assets gerados em um ambiente interativo completo e demonstrar como funcionalidades nativas do motor de jogo podem ser combinadas com o pipeline para viabilizar experiências dinâmicas sem produção artística manual.

1.5.2 Objetivos Específicos

Para o alcance do objetivo geral, o trabalho foi estruturado em torno de 8 objetivos específicos complementares, cada um correspondendo a um componente investigado e implementado na arquitetura do sistema:

OE1. Implementar um pipeline de orquestração autônomo em Python capaz de coordenar a comunicação entre modelos de linguagem, o ambiente de geração de imagens ComfyUI, a infraestrutura de persistência Supabase e o motor de jogo Godot Engine 4.5, operando em ciclos contínuos sem necessidade de intervenção humana. A orquestração centralizada é o elemento que confere ao sistema sua capacidade de operar de forma autônoma e contínua, sendo o elo entre todos os demais componentes da arquitetura. Os componentes determinísticos do pipeline, incluindo o parser de respostas LLM e o processador de remoção de fundo, são cobertos por suíte de testes unitários automatizados, com execução contínua via integração contínua no GitHub Actions.

OE2. Investigar o uso de LLMs como coordenadores semânticos do pipeline, desenvolvendo um sistema de geração de conceito que produza, em uma única requisição estruturada, o nome do inimigo e os prompts necessários para a geração de todos os ativos visuais da fase, a partir de parâmetros de tipo corporal e tipo de movimento controlados pelo próprio pipeline, garantindo coerência temática entre personagem e cenário. Essa abordagem representa uma alternativa aos sistemas procedurais tradicionais baseados em regras fixas, permitindo variabilidade criativa e contextualização temática a cada ciclo de geração.

OE3. Implementar e avaliar fluxos de geração de imagem parametrizados no ComfyUI para produção automatizada de personagens, incluindo imagem de referência e *spritesheet* de animação, e das quatro camadas de background em estilo pixel art, investigando o uso combinado de mecanismos de condicionamento visual ControlNet e IP-Adapter como estratégia para garantir consistência estética entre os ativos gerados em um mesmo ciclo.

OE4. Desenvolver e comparar métodos de remoção de fundo adaptados ao tipo de asset processado: *chroma key* adaptativo com amostragem dinâmica da cor-chave para as camadas de cenário, e segmentação semântica via modelo BRIA RMBG para o personagem, investigando as condições em que cada abordagem é mais adequada para preparar assets

com transparência para uso em motor de jogo.

OE5. Integrar os ativos gerados a uma infraestrutura de nuvem via Supabase, investigando como o desacoplamento entre geração e execução, por meio de armazenamento distribuído e interfaces serverless, viabiliza o carregamento e a montagem dinâmica das fases em tempo de execução sem que nenhum ativo esteja embutido no executável.

OE6. Implementar um sistema de objetivos procedurais gerados por LLM, com nove tipos de missão distintos e mecanismo de fallback garantido, investigando como a geração procedural de desafios pode complementar a geração visual para tornar cada fase única não apenas esteticamente, mas também em termos de gameplay.

OE7. Desenvolver um jogo funcional do gênero *beat-em-up* 2D na Godot Engine 4.5 como ambiente de demonstração do pipeline, investigando como funcionalidades nativas do motor, incluindo partículas GPU, efeitos de pós-processamento por bioma e *parallax scrolling* em quatro camadas independentes, podem ser combinadas com assets gerados proceduralmente para viabilizar experiências interativas completas sem produção artística manual.

OE8. Desenvolver um dashboard de monitoramento em tempo real para o pipeline de geração, investigando como a exposição de métricas acumuladas, eventos de progresso via WebSocket e controles operacionais, incluindo configuração do *chroma key* em tempo de execução e mecanismo de parada graciosa, contribui para a observabilidade e operabilidade de um sistema autônomo de geração contínua.

2 Metodologia

Este capítulo descreve a abordagem metodológica adotada no desenvolvimento do PhaseGen, incluindo o processo de levantamento e validação dos requisitos, a estratégia de desenvolvimento do sistema e as ferramentas utilizadas para gestão do trabalho ao longo do projeto.

2.1 Natureza da Pesquisa

O presente trabalho caracteriza-se como uma pesquisa aplicada de natureza exploratória, materializada por meio do desenvolvimento de um artefato de software funcional, o pipeline PhaseGen e o jogo que o consome. A abordagem é predominantemente prática e experimental: em vez de testar hipóteses por meio de coleta de dados controlada, o trabalho investiga a viabilidade técnica de uma arquitetura específica, a integração entre Modelos de Linguagem de Larga Escala e modelos generativos de imagem, por meio de sua implementação, operação e avaliação empírica ao longo de múltiplos ciclos de geração. Não se trata, portanto, de uma pesquisa hipotético-dedutiva no sentido estrito, com hipóteses formuladas a priori e submetidas a teste estatístico, mas de uma investigação de viabilidade técnica: o objeto de investigação é a própria arquitetura proposta, avaliada quanto à sua capacidade de operar de forma autônoma, produzir assets consistentes e sustentar execução contínua sem intervenção humana — questão diretamente refletida na formulação dos objetivos específicos da Seção 1, todos estruturados em torno do verbo "investigar".

2.2 Processo de Desenvolvimento

O desenvolvimento do sistema seguiu uma abordagem iterativa e incremental, na qual cada subsistema do pipeline, seleção de bioma, geração de conceito via LLM, síntese visual via ComfyUI, remoção de fundo, persistência no Supabase e consumo pelo cliente Godot foi implementado e testado de forma isolada antes de ser integrado aos demais componentes já funcionais. Essa estratégia permitiu identificar e corrigir falhas específicas de cada etapa, como os ajustes de threshold do *chroma key*, a substituição do mecanismo de remoção de fundo do personagem por segmentação semântica, ou a introdução do override de atributos do inimigo, sem comprometer o funcionamento das partes já validadas do sistema, e sem exigir retrabalho substancial nas etapas anteriores a cada nova integração.

O acompanhamento do andamento do desenvolvimento foi realizado por meio da ferramenta Notion, utilizada para organização de tarefas e registro do progresso ao longo das diferentes fases do projeto.

2.3 Levantamento e Validação de Requisitos

Os requisitos funcionais e não funcionais apresentados no Capítulo 3 foram inicialmente definidos antes do início da implementação, com base na literatura revisada nos Trabalhos Relacionados e nas Tecnologias Adotadas (Capítulo 1), bem como nas características técnicas das ferramentas selecionadas para compor o pipeline. Ao longo do desenvolvimento iterativo, no entanto, novas necessidades emergiram que não haviam sido previstas na especificação inicial, como a necessidade de métodos distintos de remoção de fundo por tipo de asset, ou a inclusão do dashboard de monitoramento como componente arquitetural relevante. Dessa forma, a especificação inicial cumpriu seu papel de orientar

as decisões de implementação ao longo do desenvolvimento iterativo, ao passo que a revisão final teve caráter estritamente documental, registrando como requisito formal aquilo que já havia sido decidido e implementado ao longo do processo, sem introduzir funcionalidade nova não prevista pela concepção original do projeto. Os requisitos foram revisitados e atualizados ao término do desenvolvimento para refletir com precisão o sistema efetivamente implementado, garantindo que a especificação apresentada nesta monografia corresponda ao comportamento real do PhaseGen.

A validação dos requisitos e das decisões de design foi conduzida integralmente pelo autor deste trabalho, dada a natureza individual do desenvolvimento, sem envolvimento de usuários externos.

3 Requisitos

Este capítulo apresenta os requisitos funcionais e não funcionais do sistema PhaseGen, estabelecendo as funcionalidades esperadas, as restrições arquiteturais e as características de qualidade que orientaram o desenvolvimento do projeto. A especificação aqui apresentada foi elaborada com base nos objetivos definidos no Capítulo 1 e serve como referência para as decisões de design e implementação descritas nos capítulos subsequentes.

A Tabela 4 apresenta a matriz de rastreabilidade entre os objetivos específicos definidos na Seção 1 e os requisitos funcionais e não funcionais detalhados a seguir, evidenciando o alinhamento entre a especificação e os objetivos do trabalho.

Tabela 4: Matriz de rastreabilidade entre objetivos específicos e requisitos

Objetivo	**Requisitos relacionados**
OE1	RF1, RF12, RNF5, RNF8
OE2	RF2, RF13, RNF9
OE3	RF1, RF3, RNF1, RNF7
OE4	RF3, RNF1
OE5	RF4, RF5, RNF2, RNF3, RNF4, RNF11
OE6	RF6, RF15
OE7	RF7, RF8, RF9, RF10, RNF6
OE8	RF11, RNF14

3.1 Requisitos Funcionais

Os requisitos funcionais descrevem as capacidades que o sistema deve oferecer para cumprir sua finalidade de geração autônoma de fases jogáveis. Foram identificados quinze requisitos, organizados de acordo com os subsistemas que compõem a arquitetura do PhaseGen.

RF1 — O sistema deve ser capaz de gerar automaticamente os ativos visuais de uma fase completa, incluindo *spritesheet* de animação, imagem de referência do personagem e camadas de cenário (far, mid, intermediary e near), a partir de descrições textuais produzidas proceduralmente, sem intervenção humana no processo de criação artística.

RF2 — O sistema deve produzir, por meio de uma única chamada a um modelo de linguagem, um documento estruturado contendo o nome do inimigo e os prompts de geração visual para o personagem e as camadas de cenário, a partir do tipo de movimento e do tipo corporal previamente sorteados pelo pipeline, garantindo coerência estética entre todos os ativos de um mesmo ciclo.

RF3 — O sistema deve utilizar mecanismos de condicionamento visual, ControlNet com imagens de referência de pose e IP-Adapter para preservar a identidade estética e a composição corporal do personagem entre a imagem de referência e o *spritesheet* de animação gerados sequencialmente, complementados por segmentação semântica via BRIA RMBG para remoção de fundo com preservação de detalhes de borda.

RF4 — O sistema deve armazenar os ativos visuais gerados e seus respectivos metadados em infraestrutura de nuvem, incluindo os prompts utilizados, o provedor LLM responsável, os parâmetros de geometria do *spritesheet* e as coordenadas de spawn, possibilitando rastreabilidade, reutilização e recuperação futura das informações.

RF5 — O sistema deve disponibilizar uma interface de acesso que retorne, em uma única requisição, todos os dados estruturais de uma fase, metadados da fase, camadas de background, tipo de inimigo, *spritesheet* e parâmetros de spawn em formato agregado, permitindo que o motor de jogo instancie os elementos da fase dinamicamente durante a execução.

RF6 — O sistema deve apresentar ao jogador uma interface de seleção de fases obtida dinamicamente a partir dos dados disponíveis em nuvem, permitindo a escolha manual e livre entre as fases geradas pelo pipeline.

RF7 — O motor de jogo deve carregar todos os ativos visuais, inimigos e camadas de cenário dinamicamente a partir dos dados recebidos pela interface de acesso, sem nenhum recurso embutido no executável, viabilizando composição procedural completa das fases em tempo de execução.

RF8 — O sistema deve aplicar as camadas de background com velocidades distintas de deslocamento proporcional à profundidade percebida de cada camada, criando efeito visual de profundidade por meio da técnica de *parallax scrolling*.

RF9 — O jogo deve disponibilizar mecânicas básicas de combate entre o personagem controlável e os inimigos presentes nas fases geradas.

RF10 — Os inimigos gerados devem ser instanciados dinamicamente durante a execução, com comportamento de perseguição e animações construídas em tempo de execução a partir dos metadados recuperados via interface de acesso, sem dependência de recursos pré-compilados. O inimigo detecta o jogador ao entrar em seu raio de detecção e passa a persegui-lo diretamente, transitando entre os estados de idle, chase, attack, hurt e dead conforme as condições de combate.

RF11 — O pipeline deve disponibilizar um painel de monitoramento em tempo real contendo informações sobre o status dos serviços externos, o progresso do ciclo de geração em andamento e estatísticas acumuladas de ciclos realizados, ativos produzidos e uso por provedor LLM, além de controles operacionais para iniciar, pausar e encerrar o pipeline e configuração dos parâmetros de remoção de fundo em tempo de execução.

RF12 — Todas as etapas do pipeline, seleção de bioma, geração textual, geração visual, persistência dos ativos e disponibilização para consumo devem operar de forma totalmente automatizada e integrada, sem intervenção humana após a inicialização do sistema.

RF13 — As respostas produzidas pelos modelos de linguagem devem ser obrigatoriamente validadas em formato estruturado antes de qualquer uso. Os campos de tipo de movimento e tipo corporal têm seus valores definidos internamente pelo próprio pipeline e impostos sobre a resposta do modelo, permanecendo sempre dentro dos domínios permitidos, independentemente do valor retornado pelo LLM, e garantindo interoperabilidade entre os componentes do pipeline.

RF14 — O pipeline deve evitar repetição de biomas em ciclos consecutivos, consultando as gerações mais recentes no banco de dados e excluindo os biomas já utilizados da seleção do ciclo atual.

RF15 — O histórico de fases jogadas deve ser persistido localmente e acessível por meio de uma tela dedicada, permitindo ao jogador reiniciar qualquer fase previamente selecionada.

3.2 Requisitos Não Funcionais

Os requisitos não funcionais estabelecem as restrições de qualidade, desempenho e arquitetura que o sistema deve satisfazer, independentemente das funcionalidades específicas implementadas.

RNF1 — O pipeline deve utilizar modelos generativos compatíveis com hardware de médio porte, garantindo equilíbrio entre qualidade visual e consumo computacional, de modo a viabilizar sua execução em ambientes de desenvolvimento independente sem infraestrutura de alto desempenho.

RNF2 — A disponibilização dos dados das fases ao motor de jogo deve ocorrer com baixa latência, por meio de uma interface que agregue dados de múltiplas fontes em uma única requisição, evitando chamadas sequenciais redundantes do cliente.

RNF3 — O acesso às informações persistidas deve ocorrer exclusivamente por meio de uma camada de funções serverless, sem exposição de credenciais de banco de dados ao cliente de jogo, garantindo isolamento de segurança entre os subsistemas de geração e execução.

RNF4 — A infraestrutura de backend deve suportar armazenamento distribuído de ativos visuais e persistência de metadados relacionais, com capacidade de expansão do volume de conteúdo gerado sem necessidade de alterações estruturais no banco de dados ou no esquema de armazenamento.

RNF5 — O sistema deve ser estruturado em subsistemas desacoplados em nível arquitetural, orquestração do pipeline, módulo de linguagem natural, geração de imagens, persistência em nuvem e cliente de jogo, de forma que cada subsistema possa ser substituído, atualizado ou falhar de forma isolada sem impacto direto sobre os demais.

RNF6 — O motor gráfico adotado deve oferecer suporte multiplataforma, integração nativa com serviços externos via HTTP, suporte a *spritesheet*s dinâmicos e ausência de custos de licenciamento, em conformidade com o perfil de desenvolvimento independente do projeto.

RNF7 — Os ativos produzidos pelo pipeline devem manter coerência estética e estrutural entre a imagem de referência e o *spritesheet* de animação de um mesmo inimigo, assegurada pelos mecanismos de condicionamento visual integrados ao fluxo de geração.

RNF8 — O código-fonte de cada subsistema deve seguir práticas de organização modular interna, com separação clara de responsabilidades entre classes e módulos, interfaces bem definidas entre componentes e ausência de acoplamento direto entre domínios funcionais distintos, favorecendo a manutenção corretiva e evolutiva ao longo do ciclo de desenvolvimento.

RNF9 — As respostas produzidas pelos modelos de linguagem devem ser submetidas a validação e normalização antes de qualquer uso downstream, garantindo interoperabilidade entre os componentes do pipeline mesmo em caso de respostas parcialmente não conformes com o esquema esperado.

RNF10 — O sistema deve registrar, para cada ativo gerado, os prompts utilizados, o provedor LLM responsável e os metadados de geometria pertinentes, possibilitando análise posterior e reprodutibilidade parcial das gerações.

RNF11 — O sistema deve operar de forma distribuída, com separação física entre a infraestrutura de geração de ativos, o armazenamento em nuvem e a execução do jogo, permitindo que o cliente consuma qualquer fase disponível independentemente do estado do backend de geração.

RNF12 — O mecanismo de rotação automática entre provedores de modelos de linguagem deve garantir continuidade operacional do pipeline diante de indisponibilidades

temporárias ou limites de requisições atingidos em qualquer provedor individual.

RNF13 — O pipeline deve respeitar um intervalo mínimo configurável entre ciclos de geração consecutivos, garantindo estabilização adequada dos recursos computacionais e dos serviços externos antes do início de um novo ciclo.

RNF14 — O painel de monitoramento deve receber atualizações em tempo real sobre o estado do pipeline por meio de conexão WebSocket persistente, sem necessidade de atualização manual ou *polling* pelo cliente do dashboard.

4 Design

Este capítulo apresenta as decisões de design que sustentam a arquitetura do PhaseGen. A exposição está organizada em três eixos complementares: a modelagem estrutural e comportamental do sistema por meio de diagramas UML; a visão arquitetural, que descreve os subsistemas, seus limites de responsabilidade e os mecanismos de integração entre eles; e o modelo relacional de banco de dados, que fundamenta a persistência e a distribuição dos assets gerados.

As decisões de design foram orientadas por quatro princípios centrais, diretamente derivados dos requisitos estabelecidos no Capítulo 3. O primeiro é a **autonomia operacional** do pipeline, que deve operar em ciclos contínuos sem intervenção humana após sua inicialização, exigindo mecanismos robustos de recuperação de erros, fallback entre provedores e degradação graciosa diante de falhas de componentes externos. O segundo é a **consistência visual** dos assets gerados dentro de um mesmo ciclo, garantida pelo uso combinado de mecanismos de condicionamento visual e pela estrutura sequencial do pipeline, na qual a saída de uma etapa alimenta o condicionamento da etapa seguinte. O terceiro é o **desacoplamento entre geração e consumo**: o cliente de jogo deve ser capaz de consumir qualquer fase disponível independentemente do estado do backend de geração, o que impõe a necessidade de uma camada de persistência em nuvem como intermediária entre os dois subsistemas. O quarto princípio transversal é a **segurança por isolamento e validação em camadas**: credenciais de escrita nunca são expostas ao cliente de jogo, entradas externas são validadas em múltiplos pontos independentes e o acesso aos dados persistidos ocorre exclusivamente por meio de interfaces com escopo controlado. Esse princípio manifesta-se desde a separação de credenciais entre o backend de geração e o cliente até a validação de identificadores em três camadas independentes e a restrição de origem no carregamento de assets.

A interação entre esses quatro princípios moldou as escolhas arquiteturais descritas ao longo deste capítulo, desde a separação física entre os subsistemas até as decisões de modelagem do banco de dados e a definição das interfaces de comunicação entre os componentes.

4.1 Projeto UML

A modelagem UML do PhaseGen cobre três perspectivas do sistema: funcional, estrutural e comportamental. A perspectiva funcional é representada pelos Diagramas de Caso de Uso, que delimitam os atores e as interações de alto nível com o sistema. A perspectiva estrutural é capturada pelos Diagramas de Classes, que descrevem as entidades, seus atributos, métodos e os relacionamentos estáticos entre elas. A perspectiva comportamental é coberta pelos Diagramas de Atividade, que detalham o fluxo de controle e de dados em cada subsistema. As duas categorias restantes de diagramas anunciadas anteriormente — o diagrama de componentes e o diagrama entidade-relacionamento — por descreverem, respectivamente, a organização arquitetural do sistema e o modelo relacional do banco de dados, são apresentadas junto às seções a que melhor se relacionam: o diagrama de componentes na Visão Arquitetural (Figura 9) e o diagrama entidade-relacionamento no Modelo de Banco de Dados (Figura 10).

4.1.1 Diagramas de Caso de Uso

A perspectiva funcional do PhaseGen é representada por dois diagramas de caso de uso complementares, organizados segundo os dois perfis de usuário do sistema, o Jogador e o Operador, cujos contextos de interação são suficientemente distintos para justificar representações separadas. Essa separação reflete o desacoplamento arquitetural central do sistema: o Jogador interage exclusivamente com o cliente de jogo e o Supabase como fonte de dados, sem qualquer dependência do estado do pipeline de geração; o Operador interage diretamente com o backend de geração, ComfyUI e provedores LLM, sem qualquer interação com o cliente de jogo.

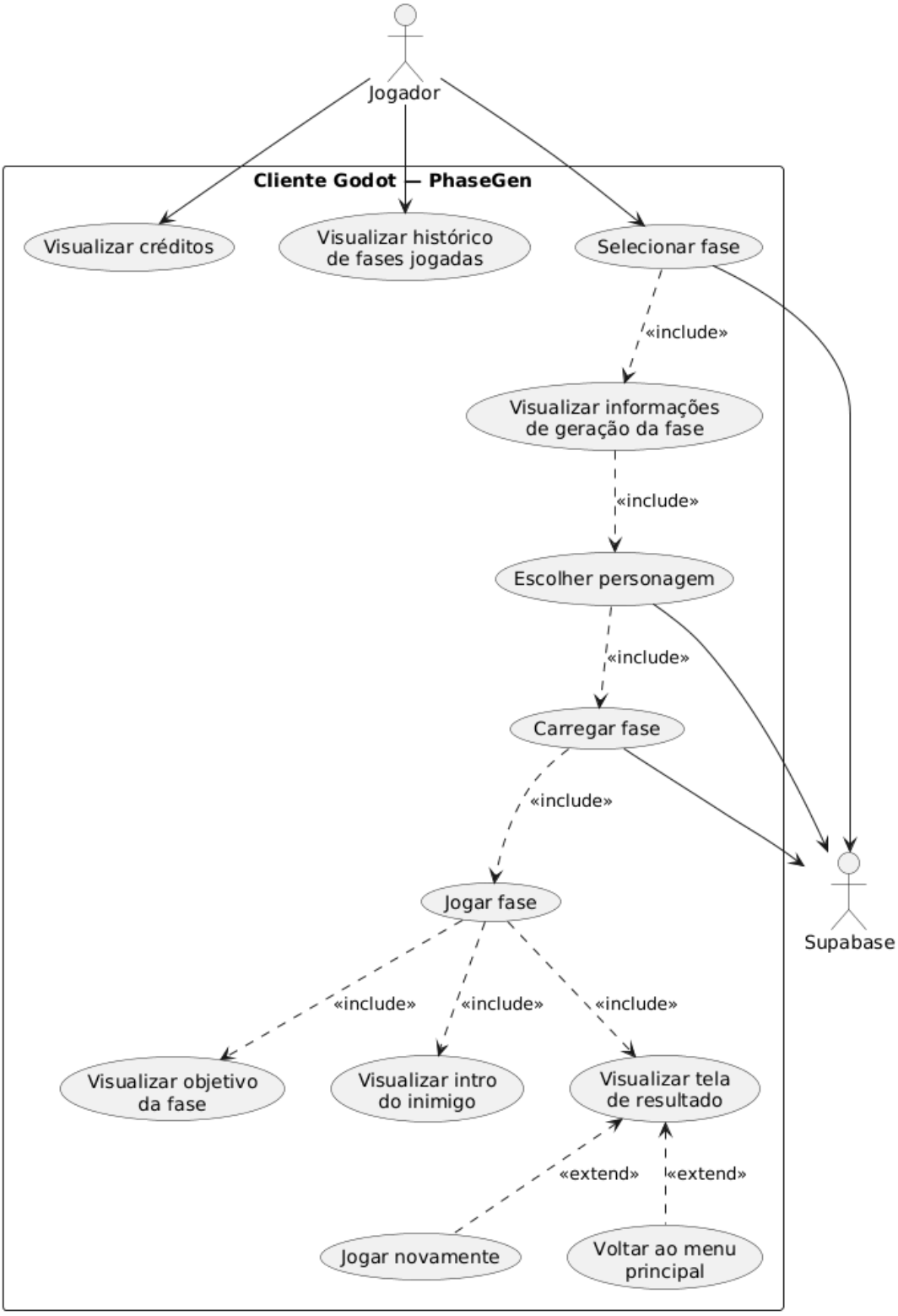

Figura 5: Diagrama de Caso de Uso: Cliente Godot

O diagrama da Figura 5 representa as interações do Jogador com o cliente de jogo. O fluxo principal é iniciado pela seleção de fase, que inclui obrigatoriamente a visualização das informações de geração da fase — contendo os prompts utilizados, o provedor LLM responsável e os metadados do ciclo — e a escolha do personagem jogável. A partir da seleção, o sistema carrega os assets dinamicamente a partir do Supabase e inicia a fase,

que inclui a visualização do objetivo procedural, a apresentação do inimigo por meio de sua fala de introdução e, ao término da partida, a exibição da tela de resultado. Esta pode ser estendida pelo jogador com as ações de jogar novamente ou retornar ao menu principal. Complementarmente, o Jogador pode acessar o histórico de partidas anteriores e a tela de créditos diretamente pelo menu principal.

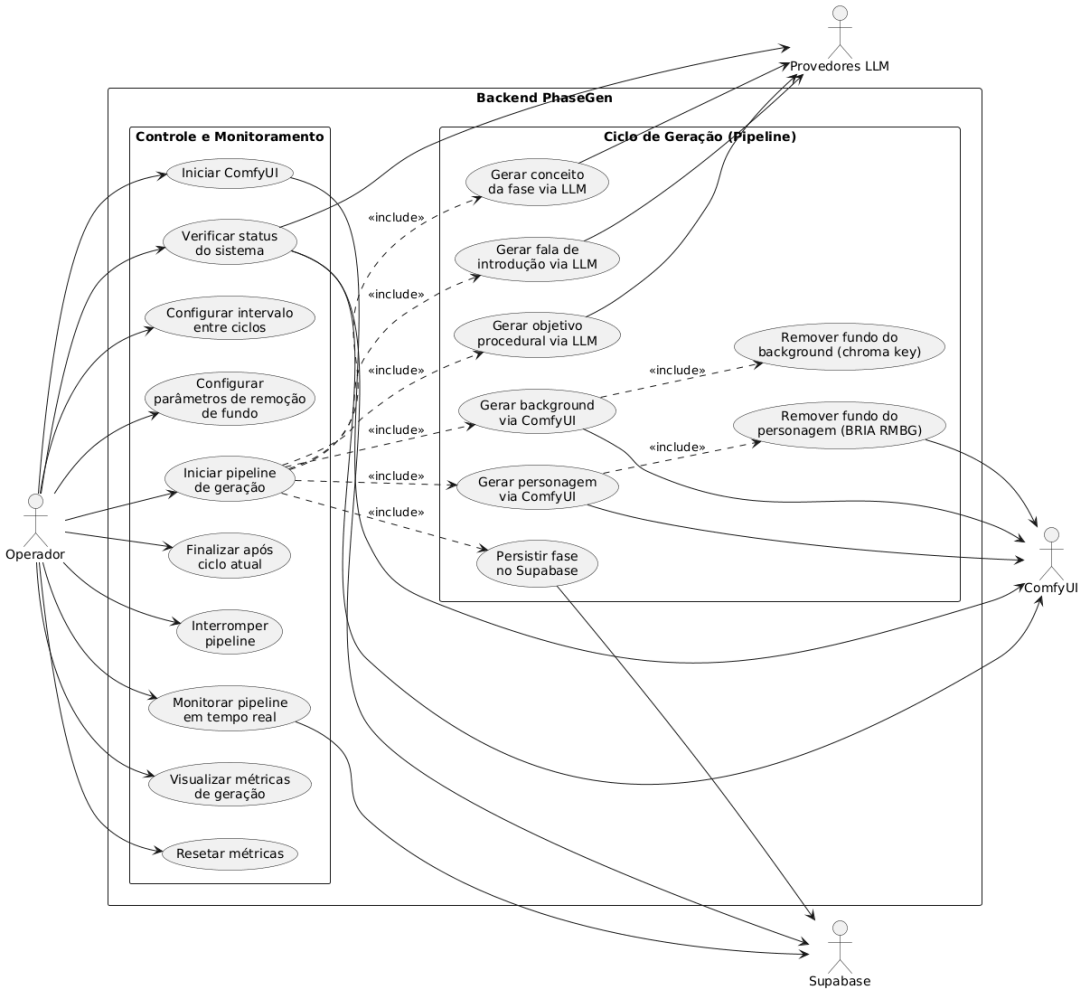

Figura 6: Diagrama de Caso de Uso: Backend PhaseGen

O diagrama da Figura 6 representa as interações do Operador com o backend de geração, organizadas em dois domínios: controle e monitoramento do sistema, e o ciclo de geração autônomo disparado pelo pipeline.

No domínio de **controle e monitoramento**, o Operador pode iniciar o ComfyUI, verificar o status dos serviços externos, ComfyUI, provedores LLM e Supabase, configurar o intervalo entre ciclos e os parâmetros de remoção de fundo, iniciar o pipeline de geração, interrompê-lo imediatamente ou solicitar sua finalização após o ciclo em andamento, e monitorar o processo em tempo real via WebSocket. O acesso às métricas acumuladas e a possibilidade de redefini-las também estão disponíveis nesse domínio.

No domínio do **ciclo de geração**, o caso de uso central é iniciar o pipeline de geração, que desencadeia um conjunto de casos de uso incluídos executados autonomamente: geração do conceito da fase via LLM, geração do objetivo procedural e da fala de introdução do inimigo via LLM, geração do personagem e do background via ComfyUI, cada um incluindo suas respectivas etapas de remoção de fundo, e persistência da fase no Supabase.

Vale destacar que, embora o diagrama represente esses casos de uso como inclusões paralelas de "Iniciar pipeline de geração", a sequência real de execução, da geração textual à persistência, está detalhada nos Diagramas de Atividade apresentados na Subseção 4.1.3.

4.1.2 Diagramas de Classes

A perspectiva estrutural do PhaseGen é representada por três diagramas de classes complementares: um para o backend Python e dois para o cliente Godot, divididos entre os serviços globais e telas de interface de um lado, e o núcleo de gameplay do outro. Essa divisão reflete a separação arquitetural real entre os subsistemas. Os pacotes descritos a seguir — orchestrator, llm, comfyui, chroma_key e storage — representam agrupamentos lógicos de responsabilidade adotados na modelagem UML, e não correspondem necessariamente à estrutura física de diretórios do repositório.[1] Cabe ainda observar que nem todos os componentes modelados como classes correspondem a classes na implementação: apenas `ComfyUIClient` e a hierarquia de provedores (`BaseProvider` e suas subclasses) são implementados como classes propriamente ditas, enquanto os demais — a exemplo de `DashboardServer`, `ChromaKey`, `MetadataWriter` e `Uploader` — correspondem a módulos baseados em funções de nível superior, representados como serviços (estereótipo `«service»`) nos diagramas para fins de clareza estrutural. Os diagramas completos estão apresentados no Apêndice B.

Backend Python

O diagrama do Apêndice B.1 ilustra a organização do backend em cinco pacotes com responsabilidades bem delimitadas.

O pacote **orchestrator** constitui o núcleo de controle do sistema. A classe `DashboardServer` expõe os endpoints REST e o canal WebSocket do servidor, gerencia o ciclo de geração autônomo e coordena todas as demais camadas do pipeline. A classe `CycleMetrics` acumula eventos de desempenho em memória durante cada ciclo, sendo consultada ao término para registro das métricas agregadas.

O pacote **llm** agrupa os componentes responsáveis pela geração de conteúdo textual. A classe abstrata `BaseProvider` define o contrato comum aos três provedores concretos, `GroqProvider`, `GeminiProvider` e `OpenRouterProvider`, permitindo sua utilização em esquema de rodízio com cooldown automático por rate limit. As classes `ConceptBuilder` e `ObjectiveBuilder` encapsulam a lógica de geração do conceito visual da fase e do objetivo procedural, respectivamente, delegando ao `Parser` a extração e validação dos documentos JSON retornados pelos modelos. Os nove tipos de objetivo suportados, representados como a enumeração `ObjectiveType` nos diagramas, são implementados de forma espelhada entre o backend e o cliente de jogo, cada subsistema declarando seu próprio conjunto de valores válidos (`VALID_TYPES`).

O pacote **comfyui** abstrai toda a comunicação com o servidor local de inferência de imagens. A classe `ComfyUIClient` expõe métodos para geração do personagem completo e das camadas de background, encapsulando internamente o ciclo de submissão de jobs, *polling* de conclusão e reinicialização do processo entre jobs para liberação de memória de vídeo.

[1] No backend, apenas prompt/ e storage/ existem como pastas; os demais componentes são módulos individuais posicionados diretamente em src/, agrupados por afinidade funcional apenas para fins de clareza da modelagem.

O pacote **chroma_key** implementa o processamento de remoção de fundo das camadas de cenário. A classe `ChromaKey` aplica o algoritmo de remoção por correspondência de cor com ajuste automático de limiar, consultando os pontos de amostragem e thresholds configurados em `ChromaConfig`.

O pacote **storage** agrupa os componentes de persistência. A classe `MetadataWriter` coordena a sequência transacional de uploads e inserções no Supabase, com rollback automático em caso de falha, e depende de `SupabaseClient` para obter o cliente autenticado de acesso ao banco de dados. A classe `Uploader` encapsula o upload de imagens para o Supabase Storage, também por meio de `SupabaseClient`. Já a classe `MetricsWriter` opera de forma independente das demais, mantendo as estatísticas acumuladas de geração em arquivo JSON local com escrita atômica, sem qualquer dependência do Supabase.

Cliente Godot — Autoloads e Interface

O diagrama do Apêndice B.2 ilustra os serviços globais e as telas de interface do cliente de jogo.

O pacote **autoloads** reúne os singletons instanciados automaticamente pela engine. A classe `Supabase` é a única interface de acesso HTTP ao backend, expondo métodos para consulta de fases exclusivamente via Edge Functions, com validação de UUID antes de qualquer requisição. A classe `CacheManager` interpõe-se entre as telas e o `Supabase`, armazenando respostas por um período configurável e evitando requisições redundantes. A classe `AssetLoader` gerencia o download assíncrono de texturas com cache em memória e allowlist de origem para proteção contra SSRF. A classe `BiomeConfig` centraliza os parâmetros visuais e sonoros de cada bioma, e `Weather` instancia os sistemas de partículas atmosféricas correspondentes. As classes `SceneTransition`, `Audio` e `IntroScreen` fornecem, respectivamente, transições de cena com fade, reprodução de efeitos sonoros e exibição da tela de apresentação do projeto.

O pacote **ui** contém as telas de interface do jogo. A classe `PhaseMenu` atua como fachada, compondo dois sub-scripts especializados: `PhaseMenuGrid`, responsável pela construção dos cards de fase e do modal de informações de geração, e `PhaseMenuCache`, responsável pelo fluxo de seleção e escolha de personagem. A classe `History` exibe o histórico local de partidas, enriquecido com metadados recuperados via `CacheManager`. A classe `PauseMenu` implementa o overlay de pausa acessível durante a gameplay. A classe `MainMenu` orquestra o menu principal com slideshow de fundo e navegação por teclado e mouse.

Cliente Godot — Phase e Entities

O diagrama do Apêndice B.3 ilustra o núcleo de gameplay do cliente de jogo.

O pacote **phase** representa o subsistema de execução de uma fase. A classe `Phase` atua como fachada, compondo cinco sub-scripts com responsabilidades distintas: `PhaseLoading` coordena o carregamento assíncrono dos assets e a exibição da textbox de introdução do inimigo; `PhaseBounds` instancia o jogador, o inimigo, o gerenciador de objetivos e o menu de pausa; `PhaseParallax` constrói e atualiza as quatro camadas de background com deslocamento proporcional à profundidade percebida; `PhaseCamera` controla o seguimento da câmera ao jogador; e `PhaseHud` exibe a tela de resultado ao término da partida. A classe `ObjectiveManager` recebe os dados do objetivo procedural gerado pelo LLM, rastreia o progresso por meio de sinais emitidos pelas entidades e determina as condições de vitória ou derrota, persistindo o resultado no histórico local. Os

tipos de objetivo por ela tratados correspondem à enumeração `ObjectiveType` já descrita, declarada no cliente por meio de seu próprio conjunto de valores válidos (`VALID_TYPES`).

O pacote **entities** contém as classes que representam os personagens do jogo. A classe `Player` implementa a lógica de movimentação, salto, ataque e recebimento de dano, emitindo sinais consumidos pelo `ObjectiveManager`. A classe `Enemy` implementa uma máquina de estados com cinco estados, idle, chase, attack, hurt e dead, com atributos de combate parametrizados pelo tipo corporal definido nos dados da fase e animações construídas dinamicamente a partir do *spritesheet* carregado via `AssetLoader`.

4.1.3 Diagramas de Atividade

Os diagramas de atividade do PhaseGen descrevem o comportamento dinâmico do pipeline de geração em dois níveis complementares: o fluxo principal do ciclo autônomo e os mecanismos de tratamento de erros e resiliência. A separação em dois diagramas reflete uma decisão deliberada de clareza, reunir ambos em um único diagrama produziria uma representação excessivamente densa e de difícil leitura.

Fluxo Principal do Ciclo

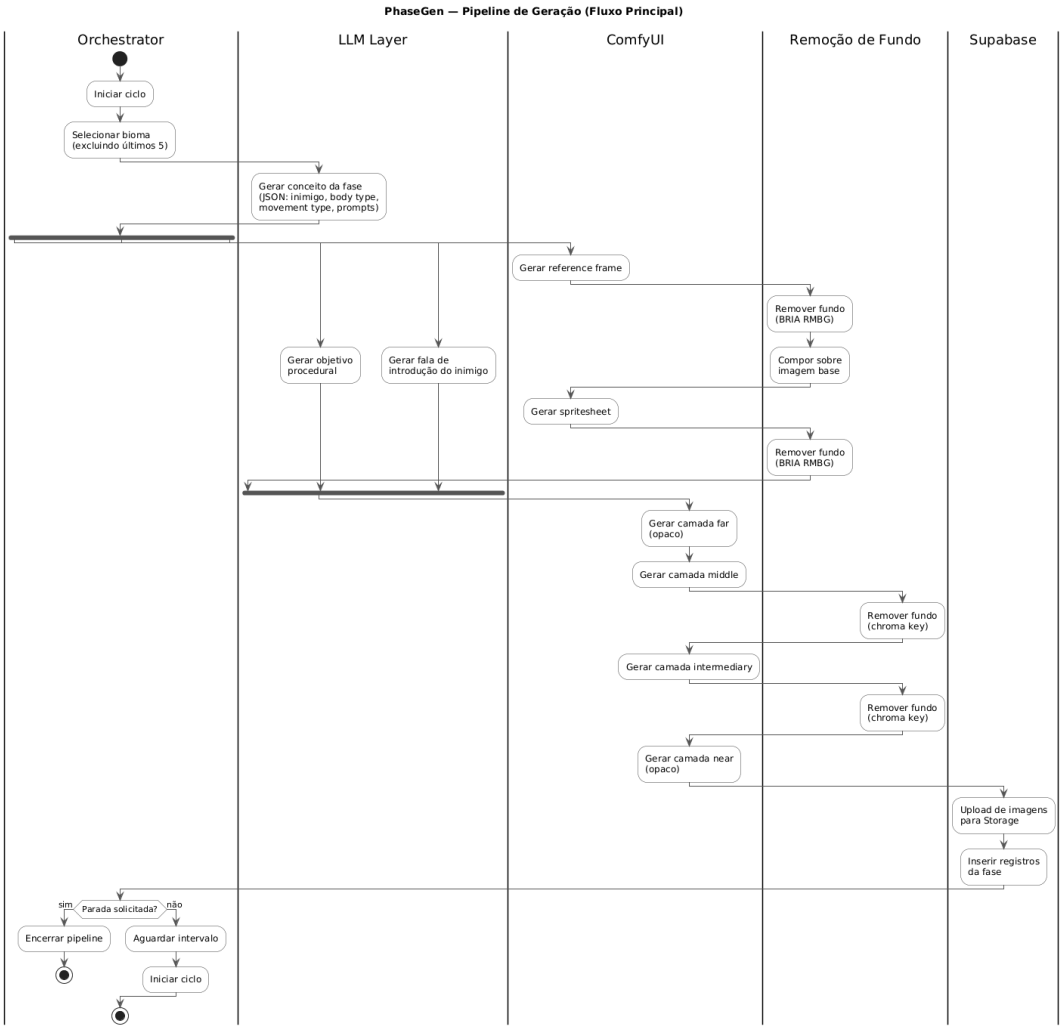

Figura 7: Diagrama de atividade: fluxo principal do pipeline de geração.

O diagrama de fluxo principal, apresentado na Figura 7, ilustra o caminho nominal de execução de um ciclo completo de geração. O ciclo tem início com a seleção do bioma, que exclui os cinco biomas gerados mais recentemente para evitar repetições consecutivas. Em seguida, uma chamada ao LLM produz o conceito estruturado da fase, nome do inimigo, tipo corporal, tipo de movimento e prompts visuais.

A etapa seguinte é executada em paralelo em três ramificações independentes: a geração do personagem, que compreende a produção do *reference frame* no ComfyUI, a remoção de fundo via segmentação semântica (BRIA RMBG), a composição sobre a imagem base e a geração da *spritesheet* com nova remoção de fundo via BRIA RMBG; a geração do objetivo procedural da fase via LLM; e a geração da fala de introdução do inimigo, também via LLM. As três ramificações são sincronizadas ao final antes do prosseguimento do ciclo.

Na etapa de geração de background, as quatro camadas são produzidas sequencialmente pelo ComfyUI. As camadas far e near não passam por remoção de fundo, pois são utilizadas como fundos opacos no jogo. As camadas mid e intermediary passam pelo processo de *chroma key* após a geração, resultando em imagens com canal alfa que permitem sobreposição transparente no motor de jogo.

Concluída a geração de todos os assets, o pipeline persiste a fase no Supabase, realizando o upload das imagens para o Storage e a inserção dos registros nas tabelas relacionais. Ao final do ciclo, o orquestrador verifica a flag de parada: se ativa, encerra o pipeline de forma controlada; caso contrário, aguarda o intervalo configurado e inicia um novo ciclo.

Tratamento de Erros e Resiliência

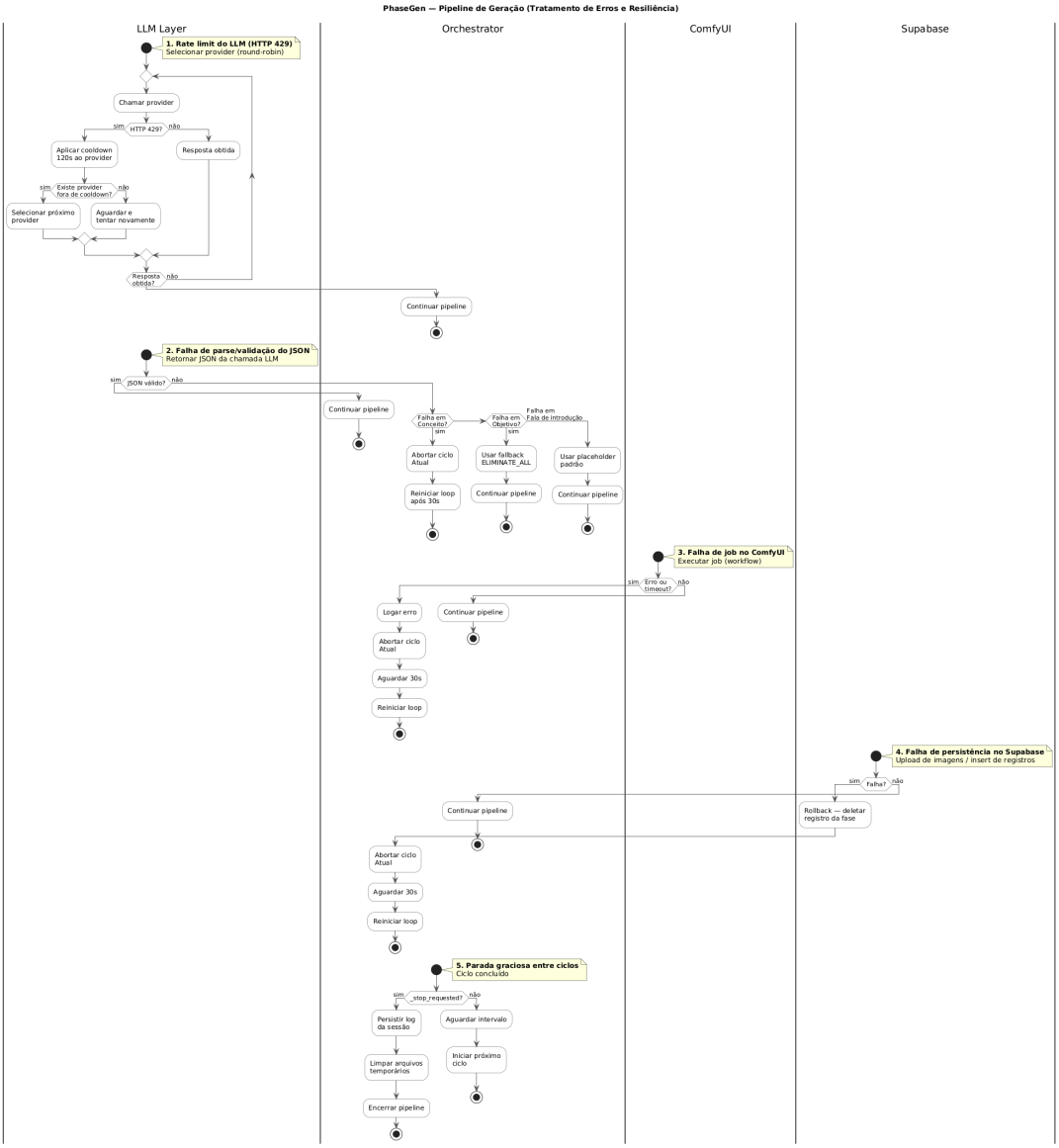

Figura 8: Diagrama de atividade: tratamento de erros e resiliência do pipeline.

O diagrama de resiliência, apresentado na Figura 8, documenta os quatro cenários de falha tratados pelo sistema e o mecanismo de parada graciosa entre ciclos.

O primeiro cenário cobre a situação de rate limit nos provedores LLM. Ao receber um erro HTTP 429, o sistema aplica um período de cooldown exclusivamente ao provedor que falhou e tenta a requisição no próximo provedor disponível em esquema de rodízio. Caso todos os provedores estejam em cooldown simultaneamente, o sistema aguarda e retenta até que um deles esteja disponível.

O segundo cenário trata falhas de parse ou validação do JSON retornado pelo LLM. O comportamento de fallback varia conforme o tipo de geração afetada: uma falha na geração do conceito da fase, o dado mais crítico do ciclo, resulta no abandono do ciclo atual e reinício após 30 segundos; uma falha na geração do objetivo resulta na adoção silenciosa do tipo padrão ELIMINATE_ALL; e uma falha na geração da fala de introdução resulta no uso de um texto substituto, sem impacto na jogabilidade.

O terceiro cenário cobre falhas de execução no ComfyUI, erros de job ou timeout. Por se tratar de uma falha em etapa central e não recuperável sem reinício, o sistema

registra o erro, abandona o ciclo atual e reinicia o loop após 30 segundos.

O quarto cenário trata falhas na persistência ao Supabase. Caso o upload de imagens ou a inserção de registros falhe, o sistema executa um rollback manual, removendo o registro de fase criado no início da transação, e abandona o ciclo atual, garantindo que nenhuma fase incompleta seja disponibilizada ao cliente.

Por fim, o mecanismo de parada graciosa é ativado quando o operador solicita o encerramento do pipeline pelo dashboard. A flag correspondente é verificada apenas entre ciclos completos, nunca durante uma execução em andamento, garantindo que todos os assets do ciclo corrente sejam gerados e persistidos antes do encerramento.

4.2 Visão Arquitetural

O PhaseGen adota uma arquitetura em pipeline composta por quatro subsistemas distintos, interligados por interfaces contratuais bem definidas: o Backend de Geração, o serviço de inferência de imagens ComfyUI, a infraestrutura de persistência e distribuição Supabase, e o cliente de jogo desenvolvido na Godot Engine 4.5. O diagrama de componentes da Figura 9 ilustra essa organização e os protocolos de comunicação entre os subsistemas, HTTP REST para a maioria das integrações, WebSocket para a transmissão de eventos em tempo real ao dashboard, e acesso exclusivo via Edge Functions para o cliente de jogo. A separação em subsistemas com responsabilidades bem delimitadas foi uma decisão arquitetural deliberada: cada componente pode evoluir, ser substituído ou falhar de forma isolada, sem impacto direto sobre os demais, o que se mostrou essencial durante o desenvolvimento iterativo do sistema. A sequência completa de execução de um ciclo, da seleção de bioma à persistência dos assets no Supabase, é detalhada no Diagrama de Atividade da Subseção 4.1.3.

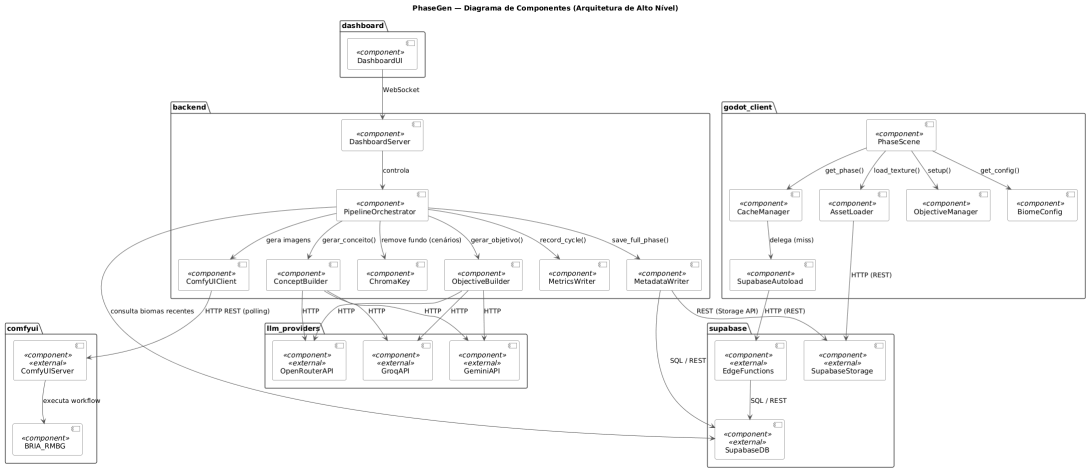

Figura 9: Diagrama de Componentes.

4.2.1 Backend de Geração

O backend de geração é o núcleo orquestrador do sistema. Implementado em Python com o framework FastAPI, ele expõe endpoints REST para controle do pipeline e um canal WebSocket para transmissão de eventos em tempo real ao dashboard de monitoramento. O ciclo de geração é executado como uma corrotina assíncrona, o que permite que operações de entrada e saída, requisições aos provedores LLM, ao ComfyUI e ao Supabase, não

bloqueiem o servidor durante a espera, preservando a responsividade do sistema mesmo durante etapas de longa duração.

A comunicação com os provedores LLM é abstraída pelo padrão de projeto Strategy, por meio de uma hierarquia de classes com contrato comum. O sistema percorre os provedores disponíveis em esquema de rodízio, aplicando período de espera exclusivamente ao provedor que retornar erro de limite de requisições, mantendo os demais disponíveis. Essa abordagem garante continuidade operacional do pipeline mesmo diante de indisponibilidades pontuais de um provedor individual, sem necessidade de intervenção humana.

4.2.2 Integração com o ComfyUI

A comunicação com o ComfyUI é realizada inteiramente via sua API REST local. O módulo cliente carrega os workflows a partir de arquivos JSON parametrizados, substitui dinamicamente os nós de prompt textual e de imagem de referência com os valores produzidos pelo módulo LLM para cada ciclo, e submete o job ao servidor para execução. O acompanhamento da conclusão é feito por *polling* periódico, executado em uma thread pool para evitar que a latência variável de inferência, tipicamente entre 30 e 80 segundos por job, bloqueie o *event loop* do servidor.

Os workflows do ComfyUI são parametrizados com o conjunto de componentes que estendem e especializam o comportamento base do Stable Diffusion 1.5 — o checkpoint ReV Animated, os adaptadores LoRA aziibpixelmix e aziib_pixel_style e o VAE OrangeMixs —, cujas funções específicas e justificativas de escolha foram apresentadas na Tabela 2. Em conjunto, esses componentes orientam a geração para o estilo pixel art pretendido e asseguram coerência estilística entre os assets produzidos em um mesmo ciclo.

A geração do personagem é dividida em dois jobs sequenciais. O primeiro produz uma imagem de referência frontal do inimigo, condicionada por imagens de controle selecionadas de acordo com o tipo corporal. O segundo utiliza essa imagem de referência como condição adicional de ControlNet para garantir consistência visual, gerando o *spritesheet* com as poses de animação. Entre os dois jobs, o processo ComfyUI é reiniciado e a memória de vídeo é liberada, evitando acumulação de estado e degradação da qualidade das imagens subsequentes; a justificativa para reiniciar o processo em vez de recorrer à API de liberação de memória é discutida adiante, nas Decisões Arquiteturais Relevantes.

4.2.3 Camada de Persistência — Supabase

A camada de persistência é responsável por duas operações complementares: o armazenamento das imagens geradas no Supabase Storage, que retorna URLs públicas para acesso direto pelo cliente; e a escrita dos metadados estruturados nas tabelas relacionais do PostgreSQL, mantendo a integridade referencial por meio das chaves estrangeiras definidas no schema.

O Supabase desempenha também o papel de camada de integração entre o backend de geração e o cliente de jogo: as Edge Functions são a única interface de leitura do cliente e permitem compor e formatar os dados de forma independente do schema interno, desacoplando o contrato público da estrutura relacional subjacente. Como o cliente nunca acessa o banco diretamente, nenhuma credencial de escrita é exposta ao lado cliente.

4.2.4 Cliente de Jogo — Godot Engine 4.5

O cliente de jogo, desenvolvido inteiramente em GDScript, não contém nenhum asset visual embutido: todos os recursos, texturas de background e *spritesheet* do inimigo, são carregados em tempo de execução a partir das URLs públicas retornadas pelas Supabase Edge Functions, interface exclusiva de acesso aos dados do Supabase a partir do cliente. Esse carregamento dinâmico é gerenciado de forma assíncrona, atualizando os elementos visuais da cena após a conclusão de cada download, sem bloquear a execução do jogo.

O efeito de *parallax scrolling* é implementado com quatro camadas de rolagem independentes, far, mid, intermediary e near, cada uma com fator de deslocamento proporcional à sua profundidade percebida, criando ilusão de profundidade característica do gênero *beat-em-up* sem necessidade de renderização tridimensional. Serviços compartilhados, como cache de texturas, transições de cena, configuração visual por bioma e reprodução de áudio, são centralizados em singletons globais instanciados automaticamente pela engine, eliminando a necessidade de passar referências entre cenas e favorecendo a manutenibilidade do código.

A ambientação visual de cada fase é complementada por efeitos nativos da engine, combinando elementos globais de estética retro com efeitos específicos configurados programaticamente de acordo com o bioma da fase carregada. Globalmente, todas as fases recebem sobreposições de grão (*grain*) e *scanlines*, reforçando uma identidade visual retro consistente em todo o jogo. O sistema de partículas GPU é utilizado para simular elementos atmosféricos característicos de cada ambiente, enquanto efeitos de pós-processamento como *glow*, névoa 2D e iluminação ambiente reforçam a identidade visual de cada bioma.

Adicionalmente, a maior parte dos biomas recebe um shader específico de ambientação, aplicado em tela cheia ou apenas na camada de *parallax* near, como distorção de calor no deserto e no vulcânico, ou refração em cristal. Alguns biomas utilizam mecanismos de shader complementares, como o efeito de oclusão de luz da caverna, enquanto outros apoiam-se exclusivamente na combinação de partículas, iluminação ambiente e neblina já descrita. Essa combinação de efeitos globais e específicos por bioma contribui para que a experiência do jogador varie não apenas nos assets gerados pelo pipeline de IA, mas também na atmosfera visual geral de cada fase.

4.2.5 Dashboard de Monitoramento

O dashboard de monitoramento constitui um componente arquitetural do sistema, servido pelo próprio backend de geração como uma página HTML estática. A comunicação em tempo real entre o servidor e o dashboard é estabelecida via WebSocket, por meio do qual o backend transmite eventos estruturados a cada mudança de estado relevante do pipeline, início de ciclo, conclusão de etapa, erros e métricas acumuladas.

O dashboard oferece três funcionalidades principais. A primeira é o monitoramento em tempo real do ciclo em andamento, com exibição do bioma selecionado, da etapa atual e do log de eventos do pipeline. A segunda é o painel de métricas acumuladas, que exibe estatísticas de todas as sessões anteriores, tempo médio por ciclo, distribuição de assets por tipo corporal e por bioma, e taxa de sucesso por provedor LLM, lidas do arquivo de métricas persistido entre sessões. A terceira é o painel de configuração do *chroma key*, que permite ajustar os parâmetros de remoção de fundo em tempo real, sem necessidade de reiniciar o servidor, com persistência automática dos valores configurados para o ciclo seguinte.

Além do monitoramento passivo, o dashboard oferece controles ativos sobre o

pipeline: iniciar e interromper o ciclo de geração imediatamente, ou solicitar encerramento após a conclusão do ciclo em andamento, garantindo que nenhum asset seja deixado em estado incompleto no banco de dados.

4.2.6 Mecanismos de Resiliência e Recuperação de Erros

A operação autônoma e contínua do pipeline impõe requisitos específicos de resiliência, uma vez que falhas em componentes externos, provedores LLM, ComfyUI e Supabase, são esperadas ao longo de sessões de longa duração. O sistema implementa quatro mecanismos complementares: o **fallback entre provedores LLM**, que redireciona a chamada para o próximo provedor do rodízio diante de erro de limite de requisições; a **recuperação automática de ciclo**, que registra falhas não esperadas, como timeouts do ComfyUI, e reinicia o loop após um intervalo fixo sem reinicialização do servidor; o **rollback de persistência**, que remove o registro de fase criado no início da operação caso a sequência de inserções falhe, evitando registros órfãos; e a **parada graciosa**, que conclui o ciclo em andamento antes de encerrar o pipeline. O tratamento específico de cada cenário de falha, incluindo os comportamentos de fallback distintos para conceito, objetivo e fala de introdução, está detalhado na Subseção 4.1.3, associado ao diagrama de atividade de tratamento de erros (Figura 8).

4.2.7 Decisões Arquiteturais Relevantes

As decisões de design mais relevantes do sistema são consolidadas nesta seção, com suas respectivas justificativas.

Amostragem adaptativa de cor no *chroma key*

A cor-chave utilizada na remoção de fundo das camadas de background é derivada por amostragem direta da imagem gerada, em vez do uso de um valor fixo predefinido. Essa decisão fundamenta-se no comportamento dos modelos de difusão, que produzem variações sutis de tonalidade do fundo entre gerações distintas. A amostragem adaptativa reduz significativamente essa variabilidade, produzindo remoções mais precisas e consistentes do que qualquer cor fixa poderia oferecer.

Métodos distintos de remoção de fundo por tipo de asset

O pipeline emprega dois métodos de remoção de fundo com aplicações distintas. Para as camadas de cenário mid e intermediary, o *chroma key* adaptativo mostra-se adequado na maioria dos casos: os fundos dessas camadas apresentam cor uniforme e bem definida, tornando a remoção por cor precisa e eficiente. Para o *reference frame* e o *spritesheet* do personagem, o *chroma key* mostrou-se inadequado: personagens humanoides gerados pelos modelos de difusão frequentemente apresentam bordas complexas, cabelos, dedos, elementos de roupa, em regiões onde a cor do fundo é próxima à cor do personagem, resultando em remoção parcial de pixels pertencentes ao sujeito. A substituição pelo modelo BRIA RMBG, integrado via custom node ComfyUI-BRIA_AI-RMBG, resolveu esse problema ao operar por segmentação semântica em vez de correspondência de cor, identificando o personagem como sujeito independentemente das características cromáticas do fundo. Para esses assets, o *chroma key* permanece implementado como opção configurável, porém desativado por padrão, delegando a remoção de fundo à segmentação semântica do BRIA RMBG.

Override de atributos do inimigo

Os campos de tipo corporal e tipo de movimento são determinados exclusivamente pelo pipeline, por sorteio aleatório realizado antes da chamada ao LLM. Os valores sorteados são injetados no prompt como contexto, orientando o modelo a produzir descrições visuais coerentes com os atributos definidos, por exemplo, prompts adequados a um inimigo de tipo corporal *heavy* com movimento *brutal*. Ao receber a resposta, o pipeline ignora os valores retornados pelo LLM nesses campos e substitui obrigatoriamente pelos valores sorteados originalmente, garantindo que o LLM nunca tenha controle efetivo sobre essas classificações.

Essa decisão fundamenta-se em testes iniciais que evidenciaram concentração expressiva das gerações em poucas combinações de atributos quando o LLM tinha liberdade para definir esses campos, comprometendo a diversidade visual dos assets ao longo do tempo.

Reinício completo do ComfyUI entre jobs

Entre os dois jobs de geração de personagem, o processo ComfyUI é encerrado e reiniciado em vez de apenas solicitar a liberação de memória via API. Essa decisão foi motivada pelo comportamento observado em hardware com memória de vídeo limitada: a API de liberação de memória do ComfyUI frequentemente deixa fragmentos residuais que causam erros de alocação no job subsequente. O reinício completo, apesar de introduzir latência adicional de 15 a 30 segundos, elimina esse problema de forma definitiva.

Rollback manual de persistência

A sequência de inserções no banco de dados é tratada como uma operação atômica por meio de rollback manual, em vez de transações SQL nativas. Essa decisão foi motivada pela limitação da biblioteca cliente utilizada, que opera sobre a API REST do Supabase e não expõe suporte a transações de banco de dados. O rollback manual replica o comportamento esperado de uma transação para o caso de uso específico do pipeline, garantindo consistência sem requerer acesso direto ao PostgreSQL.

Cache com TTL no cliente de jogo

As consultas às Edge Functions do Supabase são cacheadas no cliente por um período de cinco minutos. Essa decisão equilibra a atualidade da lista de fases disponíveis com a redução de chamadas às funções serverless, cujo limite de invocações no plano gratuito do Supabase constitui um recurso finito. O comprometimento aceito é que o menu de seleção de fases pode apresentar um atraso de até cinco minutos para refletir as fases mais recentes, o que não compromete a experiência do jogador em condições normais de uso.

4.2.8 Subsistemas e Responsabilidades

A Tabela 5 apresenta um resumo dos subsistemas, suas responsabilidades principais e as tecnologias empregadas.

Tabela 5: Subsistemas e responsabilidades

Subsistema	**Responsabilidade Principal**	**Tecnologia**
Backend de Geração	Orquestração do ciclo autônomo: seleção de bioma, requisição ao LLM com fallback, submissão de jobs ao ComfyUI, gerenciamento de memória de vídeo e persistência dos metadados	Python, FastAPI
Módulo LLM	Geração do conceito visual completo — temática do inimigo e prompts de personagem e background — em documento estruturado, a partir do bioma selecionado	Groq (Llama 3.1 8B Instant), Google Gemini (Gemini 2.0 Flash), OpenRouter
ComfyUI	Inferência de imagens em estilo pixel art via Stable Diffusion 1.5, controlada por ControlNet e IP-Adapter, por meio de workflows parametrizados via API REST local	ComfyUI, Stable Diffusion 1.5, ControlNet, IP-Adapter
Supabase	Banco de dados relacional para metadados, armazenamento de objetos para imagens geradas e Edge Functions como interface exclusiva de leitura para o cliente	PostgreSQL, Supabase Storage, Deno
Cliente de Jogo	Consumo dos assets gerados via Edge Functions, montagem dinâmica das fases e execução da sessão de jogo com mecânicas de combate, *parallax scrolling* e animação procedural de inimigos	Godot Engine 4.5, GDScript

4.3 Modelo de Banco de Dados

O modelo relacional do PhaseGen é composto por cinco tabelas implementadas no PostgreSQL gerenciado pelo Supabase. O schema foi projetado para registrar não apenas os assets gerados, mas também os metadados do processo de geração, incluindo os prompts utilizados e o provedor LLM responsável, de forma a permitir rastreabilidade e auditoria de cada ciclo.

Todas as tabelas adotam UUIDs como chaves primárias, gerados pela função nativa do PostgreSQL. Essa decisão elimina colisões em cenários de inserção distribuída, torna os identificadores opacos e não sequenciais, dificultando a enumeração de registros por clientes não autorizados e delega a geração do valor ao banco de dados, sem dependência do código da aplicação. A Figura 10 apresenta o diagrama entidade-relacionamento do schema.

Figura 10: Diagrama Entidade Relacionamento do PhaseGen.

4.3.1 Tabela phases

A tabela phases é a entidade raiz do modelo. Cada registro representa um ciclo de geração completo, identificado por um bioma e um nome gerado automaticamente com o timestamp de criação. O campo created_at é utilizado pelo pipeline para consultar os biomas gerados mais recentemente e evitar repetições em ciclos consecutivos. O campo biome possui restrição CHECK que aceita doze valores: snow, desert, forest, jungle, swamp, cave, crystal, volcanic, graveyard, ruins, mechanical e corrupted.

Dois campos merecem atenção especial. O primeiro é objective, armazenado em formato JSONB, que contém o documento completo do objetivo procedural da fase, tipo de missão e parâmetros numéricos associados. A decisão de representar o objetivo como um documento JSON em vez de uma tabela relacional separada fundamenta-se na natureza heterogênea dos nove tipos de objetivo suportados: cada tipo possui um conjunto distinto de parâmetros, alguns numéricos, outros ausentes, o que tornaria uma modelagem relacional tradicional excessivamente complexa, com muitas colunas nulas ou tabelas de atributos variáveis. O formato JSONB preserva a flexibilidade necessária para acomodar essa variabilidade sem comprometer a integridade do schema. O segundo campo é intro_speech, que armazena a fala de apresentação do inimigo gerada pelo LLM, consumida pelo motor de jogo no momento de carregamento da fase.

4.3.2 Tabela enemy_types

A tabela enemy_types registra o tipo de inimigo associado a cada fase, vinculado por chave estrangeira à tabela phases. Os campos movement_type e body_type armazenam as classificações que determinam a seleção das imagens de controle utilizadas no processo de geração visual: movement_type pode assumir os valores normal ou brutal, enquanto body_type aceita balanced, light ou heavy. Ambos os campos possuem restrições CHECK no banco de dados e defaults definidos (normal para movement_type e balanced para body_type). Os valores efetivamente inseridos são os sorteados pelo pipeline, sempre dentro dos domínios permitidos; as restrições CHECK atuam como camada adicional de integridade no nível do banco, protegendo o schema contra valores fora do domínio independentemente de sua origem.

4.3.3 Tabela sprites

A tabela sprites armazena os assets de personagem em dois tipos distintos, controlados pelo campo type com restrição CHECK: reference para a imagem de referência frontal e *spritesheet* para a folha de sprites de animação. A decisão de representar ambos os tipos em uma única tabela, em vez de tabelas separadas para cada tipo de asset, fundamenta-se na identidade estrutural dos registros: ambos compartilham os mesmos campos relevantes, diferindo apenas no valor do campo type e na URL da imagem armazenada. Essa abordagem simplifica as consultas da Edge Function, que recupera os dois assets do personagem em uma única operação, e facilita a adição de novos tipos de asset de personagem no futuro sem alteração do schema.

Os campos prompt, negative_prompt e llm_provider são armazenados exclusivamente para fins de rastreabilidade e análise do processo de geração, permitindo que cada asset seja auditado individualmente. O campo seed está reservado para versões futuras que implementem reprodutibilidade determinística de geração, não sendo preenchido na versão atual, em que o seed é gerado aleatoriamente pelo ComfyUI a cada job.

4.3.4 Tabela background_layers

A tabela background_layers registra as camadas de background de cada fase. O campo layer possui restrição CHECK que aceita exclusivamente os valores far, mid, intermediary e near, correspondendo às quatro camadas do sistema de *parallax* do jogo. Cada registro armazena a URL pública da imagem no Supabase Storage, o prompt positivo e negativo utilizados na geração e o provedor LLM responsável, mantendo a mesma filosofia de rastreabilidade adotada na tabela sprites. O campo seed, assim como naquela tabela, está reservado para versões futuras que implementem reprodutibilidade determinística de geração, não sendo preenchido na versão atual do pipeline.

A separação das camadas em registros independentes, em vez de um único registro com múltiplas URLs, foi uma decisão deliberada de design. Além de permitir que cada camada seja consultada, atualizada ou regenerada individualmente sem impacto nas demais, essa granularidade abre espaço para extensões futuras relevantes, como a possibilidade de associar variações de uma mesma camada a diferentes condições de iluminação ou períodos do dia dentro de um mesmo bioma, sem necessidade de reestruturação do schema.

4.3.5 Tabela phase_enemies

A tabela phase_enemies funciona como tabela de associação entre phases e enemy_types, formalizando o relacionamento entre uma fase e o tipo de inimigo que a compõe. Embora na versão atual do sistema cada fase contenha exatamente um inimigo, a separação entre a entidade do tipo de inimigo e sua associação à fase é uma decisão arquitetural deliberada que preserva a integridade do modelo relacional e habilita extensões relevantes sem alteração estrutural do schema: uma mesma fase poderia futuramente instanciar múltiplos inimigos de tipos distintos, e um mesmo tipo de inimigo poderia ser reutilizado em fases diferentes, sem duplicação de dados nas tabelas de origem.

Essa separação também reflete uma distinção conceitual importante do domínio: o tipo do inimigo, seus atributos visuais, nome e classificação corporal, é uma entidade independente da fase em que aparece. A tabela phase_enemies materializa essa distinção, mantendo cada responsabilidade em sua entidade correspondente.

4.3.6 Armazenamento de Objetos — Supabase Storage

O modelo relacional descrito nas seções anteriores é complementado pelo Supabase Storage, responsável pelo armazenamento das imagens geradas pelo pipeline. Todos os assets visuais, imagens de referência, *spritesheet*s e camadas de background, são persistidos em um bucket público único, configurado com política de acesso irrestrito à leitura. Essa configuração é uma decisão arquitetural deliberada: ao tornar o bucket publicamente acessível, as URLs retornadas pelo Storage no momento do upload podem ser armazenadas diretamente nos campos image_url das tabelas sprites e background_layers e consumidas pelo motor de jogo via requisições HTTP simples, sem necessidade de autenticação ou geração de URLs temporárias assinadas a cada acesso.

Essa separação entre metadados relacionais e conteúdo binário, onde o PostgreSQL armazena apenas a referência à imagem e o Storage armazena o arquivo em si, é um padrão consolidado em arquiteturas de backend modernas. No contexto do PhaseGen, ela oferece uma vantagem prática adicional: o cliente Godot pode baixar as texturas diretamente do Storage sem passar pelo pipeline de geração ou pelas Edge Functions, reduzindo a latência de carregamento e desacoplando completamente o consumo dos assets de qualquer lógica de negócio do backend.

4.3.7 Resumo do schema relacional

A Tabela 6 apresenta um resumo do schema relacional com as principais características de cada entidade.

Tabela 6: Resumo do schema relacional PhaseGen

Tabela	**Chave Primária**	**Chaves Estrangeiras**	**Campos de Destaque**
phases	id (UUID)	—	biome (CHECK), name, objective (JSONB), intro_speech, created_at
enemy_types	id (UUID)	phase_id → phases (CASCADE)	name, body_type (CHECK), movement_type (CHECK)
sprites	id (UUID)	enemy_type_id → enemy_types (CASCADE)	type (CHECK), image_url, prompt, negative_prompt, llm_provider, seed
background_layers	id (UUID)	phase_id → phases (CASCADE)	layer (CHECK), image_url, prompt, negative_prompt, llm_provider, seed
phase_enemies	id (UUID)	phase_id → phases (CASCADE), enemy_type_id → enemy_types (CASCADE)	—

As restrições CHECK nas tabelas sprites, background_layers, enemy_types e phases constituem uma camada adicional de integridade referencial: mesmo que o código da aplicação falhe em validar os valores desses campos, o banco de dados rejeita inserções fora dos domínios permitidos. Em um pipeline autônomo onde respostas inesperadas do LLM são uma possibilidade real, essa dupla camada de validação, no código da aplicação e no schema do banco de dados, é uma decisão de design relevante para a robustez do sistema. Essa mesma filosofia de resiliência, presente ao longo de toda a arquitetura do PhaseGen, manifesta-se aqui na forma de um schema que protege a integridade dos dados independentemente do comportamento dos componentes externos que o alimentam.

5 Testes de Software

5.1 Projeto de Testes

A estratégia de testes do PhaseGen combina testes unitários automatizados para os componentes determinísticos do pipeline e testes funcionais em ambiente real para os subsistemas dependentes de hardware e serviços externos. Essa divisão reflete a natureza heterogênea do sistema: componentes como o parser de respostas LLM, o processador de *chroma key* e a camada de persistência possuem lógica determinística testável de forma isolada por meio de simulação de suas dependências, ao passo que a geração de imagens via ComfyUI, a comunicação real com provedores LLM e a persistência efetiva no Supabase dependem de infraestrutura específica que não pode ser simulada com fidelidade total em ambiente controlado.

Os testes unitários foram implementados com o framework pytest e organizados em nove arquivos dentro do diretório `backend/tests/`. A suíte é composta por 140 testes independentes, sem dependência de GPU, ComfyUI, Supabase ou rede, com tempo de execução total de aproximadamente 1,5 segundos. A execução automática da suíte é garantida por um workflow de integração contínua configurado no GitHub Actions, disparado a cada push ou pull request que altere arquivos do backend, utilizando Python 3.10 em ambiente Linux (`ubuntu-latest`), o mesmo interpretador adotado no ambiente de desenvolvimento local. A suíte não foi instrumentada com ferramentas de medição de cobertura de código (*statement* ou *branch coverage*) nem de *mutation testing*, o que impede quantificar objetivamente a completude da suíte além da contagem de casos de teste e dos módulos cobertos. A instrumentação dessas métricas é apontada como melhoria futura do processo de qualidade do projeto, discutida na Seção 8.1.

Os testes funcionais cobrem os subsistemas mais críticos do pipeline que não se prestam à automação isolada, organizados em três frentes complementares: integração com o ambiente de geração de imagens, persistência e acesso aos dados em ambiente real, e comportamento do cliente de jogo. Em razão da dependência de serviços externos e de hardware específico, esses testes foram conduzidos em ambiente real de execução, sem uso de mocks ou simuladores, priorizando a validação do comportamento integrado do sistema em condições próximas às de operação normal.

5.1.1 Testes Unitários

A suíte de testes unitários cobre nove módulos do backend Python, selecionados por possuírem lógica determinística passível de validação isolada por meio de simulação de dependências externas.

O arquivo `test_parser.py` contém 15 testes sobre o módulo de parsing de respostas LLM, verificando o comportamento do parser diante de JSON válido retornado diretamente, JSON embutido em blocos de markdown, JSON com campos ausentes ou malformados, e tentativas de injeção de prompt por meio dos termos bloqueados pela denylist do sistema.

O arquivo `test_concept_normalization.py` contém 35 testes sobre a normalização e validação do documento de conceito da fase, incluindo a verificação de comprimento mínimo dos prompts visuais, rejeitando descrições com menos de 20 caracteres, e a normalização dos campos `body_type` e `movement_type` para os domínios de valores permitidos. Os testes confirmaram que a ausência desses campos na resposta do LLM resulta em ValueError controlado, nunca em KeyError não tratado — confirmando que o documento de conceito exige esses campos por completude estrutural, ainda que seus valores sejam

posteriormente sobrescritos pelo mecanismo de override descrito nas Decisões Arquiteturais.

O arquivo `test_chroma_key.py` contém 13 testes sobre o processador de remoção de fundo por correspondência de cor, utilizando imagens sintéticas geradas com NumPy e Pillow em diretórios temporários. Os testes verificam a convergência do algoritmo de auto-threshold para o intervalo-alvo de pixels removidos, a remoção correta de fundos de cor uniforme, a preservação de pixels pertencentes ao personagem e o comportamento do método de amostragem da cor-chave, confirmando que esta é derivada por amostragem direta da imagem gerada, não de um valor fixo predefinido.

O arquivo `test_provider_rotation.py` contém 11 testes sobre o mecanismo de rotação entre provedores LLM, utilizando providers simulados via `unittest.mock`. Os testes verificam três garantias do sistema: a seleção do próximo provedor disponível após erro de rate limit ou erro genérico; a aplicação do período de cooldown exclusivamente ao provedor que falhou, com retorno à elegibilidade após expiração verificada por meio de relógio mockado; e o comportamento do sistema quando todos os provedores estão simultaneamente em cooldown, levantando exceção no `ConceptBuilder` e retornando o objetivo padrão `ELIMINATE_ALL` no `ObjectiveBuilder`.

O arquivo `test_biome_validation.py` contém 28 testes sobre a validação de biomas em duas camadas independentes. A primeira camada ocorre no orquestrador, no momento da seleção do bioma do ciclo, que exclui do conjunto `VALID_BIOMES` os biomas gerados recentemente. A segunda camada ocorre no próprio `ConceptBuilder`: a validação do ConceptBuilder rejeita biomas fora do domínio permitido antes de montar o prompt ou instanciar qualquer provedor, levantando `ValueError` com mensagem indicando o valor recebido e os valores aceitos. Os testes confirmam, por meio de mocks, que nenhum provider é chamado quando um bioma inválido é fornecido, e que a falha ocorre antes do avanço do índice de rodízio entre provedores.

O arquivo `test_comfyui_node_lookup.py` contém 12 testes sobre a identificação de nós nos workflows do ComfyUI por inspeção estrutural de padrões, utilizando workflows JSON sintéticos. Os testes cobrem o caminho de identificação bem-sucedida dos nós de imagem de referência e de *spritesheet*, a ausência de nós esperados, workflows vazios e estruturas inesperadas, como nós do workflow com campos obrigatórios ausentes. Os testes revelaram que essas funções não levantam exceção quando um nó esperado não é encontrado, retornando string vazia nesse caso; a validação e o tratamento de erro correspondente ficam a cargo do código chamador, comportamento que os testes documentam e verificam explicitamente.

O arquivo `test_bg_ref_selection.py` contém 8 testes sobre o mecanismo de seleção anti-repetição das imagens de referência de background, implementado no mecanismo de seleção anti-repetição de referências de background da classe `ComfyUIClient`. Os testes verificam que a seleção evita reutilizar referências empregadas em ciclos recentes e que o mecanismo retorna a uma seleção válida quando o conjunto de referências disponíveis se esgota, assegurando variabilidade visual entre as camadas de cenário de fases consecutivas.

O arquivo `test_metadata_rollback.py` contém 7 testes sobre o mecanismo de rollback da camada de persistência, com o cliente Supabase e a função de upload de imagens simulados via `unittest.mock` e arquivos locais sintéticos. Os testes verificam que falhas em diferentes pontos da sequência de persistência, upload de imagem, inserção de `enemy_types`, de `sprites` ou de `background_layers`, disparam exatamente uma chamada de rollback com o identificador de fase correto, que o caminho de sucesso não aciona nenhum rollback, e que a validação local de existência dos arquivos ocorre antes de qualquer chamada ao Supabase, evitando operações remotas desnecessárias em caso de arquivos

ausentes.

O arquivo `test_metrics_writer.py` contém 11 testes sobre a persistência de métricas, com o diretório de métricas redirecionado para um caminho temporário. Os testes verificam a normalização dos nomes de classe dos provedores para as chaves utilizadas no arquivo de métricas, a escrita atômica via arquivo temporário sem deixar artefatos órfãos, o comportamento de leitura diante de arquivo de métricas ausente ou corrompido, retornando uma estrutura padrão sem propagar exceção, e o preenchimento automático de chaves ausentes em arquivos de métricas de versões anteriores do schema.

5.1.2 Testes do Módulo de Geração Textual em Ambiente Real

Complementarmente aos testes unitários, o módulo de geração de conceitos via LLM foi validado em ambiente real por meio de execução completa do fluxo de geração para diferentes biomas, com chamadas efetivas aos três provedores configurados. Essa validação cobre aspectos que os testes unitários, por dependerem de providers simulados, não capturam: a latência real de resposta de cada provedor, o comportamento da rotação diante de indisponibilidades genuínas de rede ou de cota, incluindo os períodos de falha do Gemini registrados nas métricas de desempenho, e a qualidade semântica efetiva dos prompts retornados pelos modelos em produção, que não pode ser avaliada por mocks determinísticos.

5.1.3 Testes de Integração com o ComfyUI

A integração com o ambiente de geração de imagens foi validada por meio de testes funcionais de geração completa de assets, nos quais o pipeline executa os seis workflows em sequência para um conjunto de parâmetros conhecidos. Os aspectos verificados incluem a correta parametrização dos workflows com os prompts e imagens de referência produzidos pelo módulo de geração textual, a conclusão bem-sucedida de todos os jobs dentro dos timeouts configurados e a presença dos arquivos de saída nos caminhos esperados ao término de cada job.

A seleção das imagens de controle de pose foi verificada para todas as combinações possíveis de tipo corporal e tipo de movimento, assegurando que o mecanismo de seleção opera corretamente em todos os cenários suportados pelo pipeline. O procedimento de reinicialização do ComfyUI entre os jobs de geração de personagem foi validado confirmando que o ambiente retorna ao estado responsivo antes da submissão do job subsequente, sem erros de alocação de memória de vídeo.

5.1.4 Testes do Sistema de Remoção de Fundo do Personagem

Diferentemente do *chroma key*, cuja lógica determinística é coberta pela suíte de testes unitários, a remoção de fundo do *reference frame* e do *spritesheet* via segmentação semântica (BRIA RMBG) depende de um modelo de aprendizado profundo executado dentro do ComfyUI, não sendo passível de teste unitário isolado. Sua validação foi conduzida por inspeção visual sistemática de imagens processadas ao longo de múltiplos ciclos de geração, avaliando a preservação de detalhes finos de borda, cabelos, extremidades e elementos de vestuário, em personagens de diferentes tipos corporais e biomas, e comparando os resultados com os arquivos originais preservados no diretório de saída do ComfyUI. Cabe ressaltar que essa avaliação foi conduzida exclusivamente pelo autor deste trabalho, o que introduz viés subjetivo ao julgamento de qualidade estética e estrutural dos assets;

a ausência de avaliação por terceiros, designers ou desenvolvedores de jogos, constitui limitação metodológica discutida na Seção 5.3. Essa inspeção confirmou a superioridade do BRIA RMBG em relação ao chroma key para esse tipo de asset, conforme discutido nas Decisões Arquiteturais do Capítulo 4.

5.1.5 Testes de Persistência e API em Ambiente Real

Complementarmente aos testes unitários de rollback, a camada de persistência foi validada em ambiente real verificando que, ao término de um ciclo completo de geração, o banco de dados contém os registros esperados em todas as tabelas, fase, tipo de inimigo, assets de personagem e camadas de background, com todas as relações de chave estrangeira corretamente estabelecidas, e que as URLs dos assets persistidos são publicamente acessíveis.

As Edge Functions foram testadas por meio de chamadas HTTP diretas aos endpoints de listagem e de recuperação individual de fase, realizadas após a conclusão de ciclos de geração completos. As verificações contemplaram a estrutura do documento JSON retornado e a presença de todos os campos necessários para a instanciação dinâmica dos elementos da fase pelo cliente de jogo.

5.1.6 Testes do Cliente de Jogo

O cliente Godot foi testado de forma funcional, com execução do jogo em ambiente real e verificação dos comportamentos esperados em cada etapa do fluxo de carregamento e execução de uma fase. Os aspectos avaliados incluem o carregamento correto da lista de fases disponíveis via interface de acesso, a montagem completa de uma fase selecionada, incluindo download e exibição das camadas de background e do *spritesheet* do inimigo, o funcionamento do efeito de *parallax scrolling* com as velocidades configuradas por camada, e a execução correta das animações do inimigo com os parâmetros recuperados do banco de dados.

O mecanismo de cache de texturas foi verificado confirmando que requisições subsequentes a assets já baixados durante a sessão são atendidas localmente, sem nova comunicação com a infraestrutura de nuvem, reduzindo a latência de carregamento em fases acessadas mais de uma vez.

5.2 Métricas de Desempenho Observadas

O dashboard de monitoramento do pipeline acumula estatísticas de execução ao longo das sessões de geração, permitindo a análise quantitativa do desempenho do sistema em condições reais de operação. As métricas apresentadas a seguir foram coletadas ao longo de 96 ciclos completos de geração, executados em ambiente Windows com GPU de 8 GB de VRAM e conexão de internet estável.

O tempo médio por ciclo completo foi de aproximadamente 45 minutos (2696 segundos), sendo a geração de imagens via ComfyUI a etapa de maior duração, em especial o *spritesheet*. Esse valor reflete o custo computacional acumulado dos seis jobs de inferência executados por ciclo, *reference frame*, *spritesheet* e quatro camadas de background, além do tempo de reinicialização do ComfyUI entre os jobs de personagem. As chamadas aos provedores LLM, executadas em paralelo com parte da geração de imagens, contribuem com impacto marginal no tempo total, conforme esperado pela arquitetura de paralelismo implementada.

A distribuição por tipo corporal manteve-se aproximadamente uniforme ao longo dos 96 ciclos observados, heavy (35), light (34) e balanced (27), confirmando empiricamente a eficácia do mecanismo de override de atributos do inimigo descrito nas Decisões Arquiteturais do Capítulo 4.

A taxa de falha dos provedores LLM variou significativamente entre os provedores. O Groq apresentou desempenho estável ao longo dos ciclos. O OpenRouter registrou taxa de falha de 38%, atribuída principalmente a indisponibilidades pontuais dos modelos gratuitos utilizados. O Gemini apresentou 100% de falhas no período observado, decorrentes de problemas de integração com cadeias de redirecionamento não resolvidas — que não foram sanados dentro da janela de coleta das métricas. Durante esse período, a cota agregada efetivamente disponível ao pipeline ficou restrita a dois provedores, Groq e OpenRouter; ainda assim, o mecanismo de rotação automática garantiu continuidade operacional em todos esses cenários, sem necessidade de intervenção manual.

5.3 Limitações Identificadas

A execução dos testes permitiu identificar limitações relevantes do sistema, cujo registro contribui para a avaliação crítica da abordagem proposta.

A principal limitação observada diz respeito à qualidade da remoção de fundo das camadas de background via *chroma key*. Embora o mecanismo de amostragem adaptativa da cor-chave reduza significativamente as falhas em relação ao uso de uma cor fixa, gerações ocasionais com fundos de tonalidade muito próxima à cor de elementos do cenário, situação pouco frequente, restrita a determinadas combinações de bioma, ainda resultam em remoção parcial de pixels pertencentes ao conteúdo visual da camada. Quando ocorre, esse problema é agravado em regiões de borda, onde a compressão de imagem introduz artefatos que dificultam a distinção entre fundo e conteúdo.

Uma segunda limitação decorre da natureza estocástica dos modelos de linguagem utilizados na geração de conceito, objetivo e fala de introdução. Embora o sistema empregue validação estrutural e normalização dos campos retornados, os LLMs ocasionalmente produzem respostas com incoerências semânticas, descrições que não correspondem ao bioma selecionado, ou prompts genéricos que não exploram adequadamente o tema do inimigo, sem que essas incoerências sejam detectáveis por validação de schema, uma vez que o documento retornado permanece estruturalmente válido. Esse comportamento é consistente com a literatura sobre confiabilidade de saída estruturada em LLMs, que aponta variação significativa de consistência semântica entre modelos e configurações, mesmo quando a validade estrutural é mantida [29]. No contexto do PhaseGen, esse fenômeno não compromete a execução do pipeline, que segue operando normalmente com o conteúdo gerado, mas pode ocasionalmente resultar em fases com menor coerência temática entre o inimigo, o cenário e o objetivo proposto.

Uma terceira limitação diz respeito à relação entre a capacidade de processamento gráfico disponível, o tempo de geração e a qualidade dos assets produzidos. O ambiente de desenvolvimento e testes utilizou uma GPU AMD Radeon RX 580 8GB, executada via DirectML, camada de compatibilidade para GPUs não-NVIDIA no Windows, em vez de aceleração CUDA nativa. Essa configuração foi mantida deliberadamente ao longo de todo o desenvolvimento como cenário de pior caso (*worst-case scenario*), com o objetivo de validar a viabilidade do pipeline sob as condições de hardware mais restritivas plausíveis para um desenvolvedor independente, em vez de otimizar para o melhor desempenho possível. Essa escolha metodológica é reforçada pelo fato de que o DirectML, no ambiente

utilizado, sequer reconheceu corretamente a capacidade real de VRAM da GPU, reportando apenas uma fração do total disponível, uma limitação de compatibilidade característica do ecossistema de modelos de difusão baseados em PyTorch, historicamente construído em torno da plataforma CUDA da NVIDIA. O tempo médio de 45 minutos por ciclo, relatado nas métricas de desempenho, reflete diretamente essa configuração.

Alternativas de aceleração para GPUs AMD no Windows existem e vêm amadurecendo, embora com cobertura de hardware ainda limitada. O ROCm, plataforma oficial de computação da AMD, passou a oferecer suporte nativo ao Windows a partir de sua série 7, mas sua cobertura oficial de placas permanece restrita às arquiteturas mais recentes, RDNA3 e RDNA4, não contemplando a arquitetura Polaris da GPU utilizada neste trabalho [30, 31]. Para GPUs dessa geração, o ZLUDA, camada de tradução de instruções CUDA para hardware AMD, permanece a alternativa mais viável, com benchmarks estruturados conduzidos pela Puget Systems demonstrando desempenho comparável ou superior ao ROCm nativo em determinados workflows de geração de imagem [32]. Isso indica que o tempo de ciclo poderia ser reduzido em uma configuração otimizada com ZLUDA, sem alterar a arquitetura do pipeline, ainda que a ausência de suporte oficial para arquiteturas mais antigas evidencie que a fragmentação do ecossistema de aceleração para GPUs AMD permanece uma barreira real para desenvolvedores com hardware legado. A opção deliberada pelo cenário de hardware mais restritivo reforça, assim, a validade do PhaseGen como alternativa viável mesmo fora das condições ideais, com margem conhecida e documentada de otimização para quem dispuser de tempo para configurar uma rota de aceleração alternativa.

Uma quarta limitação diz respeito à ausência de avaliação comparativa direta entre o pipeline proposto e um fluxo de produção artística tradicional em termos de tempo e custo de produção. As afirmações sobre redução de esforço e de gargalo de produção apresentadas neste trabalho fundamentam-se na literatura revisada nos Trabalhos Relacionados e na demonstração de viabilidade técnica do pipeline, não em um experimento controlado com medição direta de produtividade comparada. A ausência desse experimento é discutida como direção de pesquisa futura na Seção 8.1.

Uma quinta limitação refere-se à avaliação de qualidade dos assets visuais gerados, conduzida integralmente pelo autor deste trabalho, conforme descrito na Seção 5. A ausência de avaliação por terceiros, designers, desenvolvedores de jogos ou usuários finais, introduz viés subjetivo ao julgamento de qualidade estética e estrutural, sendo igualmente apontada como direção de pesquisa futura.

Por fim, a avaliação da qualidade visual dos assets não empregou métricas quantitativas de similaridade ou fidelidade perceptual, como SSIM, LPIPS ou FID, tampouco CLIP Score para aderência semântica entre prompt e imagem gerada. A ausência dessas métricas deve-se à inexistência de um conjunto de referência (*ground truth*) contra o qual comparar as gerações, dado que os assets são inéditos e gerados proceduralmente, sem equivalente artesanal para servir de baseline, o que também é apontado na seção seguinte.

As seis limitações identificadas, variabilidade na remoção de fundo, incoerência semântica ocasional na geração textual, sensibilidade do tempo de ciclo à configuração de hardware, ausência de comparação experimental com um fluxo de produção tradicional, avaliação de qualidade conduzida por um único avaliador e ausência de métricas quantitativas de qualidade de imagem, não comprometem a viabilidade demonstrada do pipeline, mas delimitam com precisão o estágio atual da tecnologia e os pontos que mais se beneficiariam de investigação adicional em trabalhos futuros sobre o tema.

6 Implantação

6.1 Projeto de Implantação

A implantação do PhaseGen envolve dois ambientes distintos e independentes: o ambiente de geração, composto pelo backend Python e pelo serviço de inferência ComfyUI, executado localmente na máquina do operador; e o ambiente de consumo, composto pelo cliente de jogo Godot, que depende exclusivamente de conectividade de rede para acessar a infraestrutura em nuvem. Essa separação física entre geração e consumo é uma consequência direta da decisão arquitetural de utilizar o Supabase como camada intermediária: o cliente de jogo pode ser executado em qualquer máquina com acesso à internet, independentemente do estado ou da disponibilidade do ambiente de geração.

A Figura 11 ilustra a distribuição física dos componentes do sistema entre os dois ambientes e a infraestrutura em nuvem, incluindo os protocolos de comunicação utilizados em cada integração, HTTP e WebSocket para o dashboard local, REST para a comunicação com o ComfyUI e com os provedores LLM, e HTTPS para todo o tráfego com o Supabase.

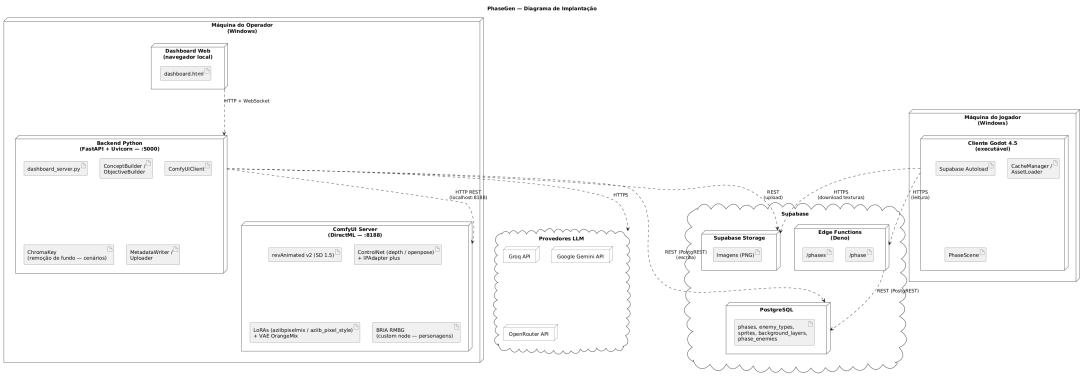

Figura 11: Diagrama de Implantação do PhaseGen.

6.1.1 Requisitos de Hardware

O ambiente de geração impõe os requisitos de hardware mais significativos do sistema, determinados principalmente pelas demandas do processo de inferência do Stable Diffusion 1.5 via ComfyUI. A Tabela 7 compara os requisitos oficiais recomendados pelas soluções utilizadas no pipeline com a configuração efetivamente empregada no desenvolvimento e validação do PhaseGen.

Tabela 7: Requisitos de hardware: recomendação oficial vs. configuração utilizada no projeto

Componente	**Requisito Oficial Recomendado**	**Configuração Utilizada no Projeto**
GPU / Aceleração	8 GB de VRAM com aceleração nativa (CUDA em GPUs NVIDIA, ou ROCm/ZLUDA em GPUs AMD)	AMD Radeon RX 580 (8 GB físicos, 1 GB reconhecido) via DirectML, sem aceleração nativa
Sistema Operacional	Windows ou Linux, com melhor desempenho documentado em Linux	Windows
RAM	16 GB	15,87 GB
Armazenamento	SSD, com espaço para checkpoint, ControlNet, IP-Adapter, LoRAs, VAE e BRIA RMBG (aproximadamente 5 a 7 GB no total)	SSD, mesma configuração

Por decisão metodológica, o sistema foi desenvolvido e validado na configuração mais restritiva apresentada na tabela, com o objetivo de testar o pipeline sob o cenário de hardware mais modesto plausível para um desenvolvedor independente, em vez de otimizar para o melhor desempenho disponível. O sistema completou todos os ciclos de geração com sucesso nessa configuração, ainda que com tempo médio de ciclo mais elevado, aproximadamente 45 minutos, conforme discutido nas Limitações Identificadas da Seção 5.3. Essa validação reforça a viabilidade do PhaseGen mesmo em condições de hardware menos favoráveis que as recomendadas, com GPUs NVIDIA ou GPUs AMD com aceleração nativa tendendo a apresentar tempos de ciclo substancialmente menores.

O requisito de RAM contempla a execução simultânea do servidor FastAPI, do processo ComfyUI com os modelos carregados, do sistema operacional e dos processos auxiliares do ambiente de desenvolvimento. O uso de SSD para o armazenamento local é recomendado para reduzir o tempo de carregamento dos modelos entre sessões e entre os jobs de geração.

O ambiente de geração requer conectividade de internet estável para comunicação com os provedores LLM e com a infraestrutura Supabase para upload de assets e inserção de metadados. O cliente de jogo requer apenas conectividade de internet para acesso às Edge Functions e ao Supabase Storage, sem demandas de hardware além das exigidas pela Godot Engine para execução de jogos 2D.

6.1.2 Requisitos de Software — Ambiente de Geração

A implantação do ambiente de geração requer os seguintes componentes de software:

Python 3.10 com as dependências do pipeline instaladas, incluindo FastAPI, Uvicorn, a biblioteca Pillow para processamento de imagens, NumPy para as operações vetorizadas do *chroma key*, e os clientes HTTP das APIs dos provedores LLM e do Supabase.

ComfyUI instalado localmente com os seguintes modelos e custom nodes posiciona-

dos nos diretórios correspondentes da instalação: o checkpoint ReV Animated, os modelos ControlNet control_v11f1p_sd15_depth e control_v11p_sd15_openpose, o modelo IP-Adapter, os adaptadores LoRA aziibpixelmix e aziib_pixel_style, o VAE OrangeMixs, e o custom node ComfyUI-BRIA_AI-RMBG para segmentação semântica do personagem. Todos os modelos devem ser obtidos e posicionados manualmente pelo operador antes da primeira execução, pois não há script de instalação automatizada para esses arquivos.

Credenciais de acesso aos serviços externos, configuradas em arquivo de variáveis de ambiente: chaves de API dos provedores Groq, Google Gemini e OpenRouter; e as credenciais do projeto Supabase, incluindo a URL do projeto e a chave de serviço utilizada pelo backend para operações de escrita no banco de dados e no Storage.

As dependências de desenvolvimento listadas em `requirements-dev.txt`, que incluem o framework pytest, são necessárias apenas para execução da suíte de testes unitários, detalhada no Capítulo 5.

6.1.3 Requisitos de Software — Infraestrutura em Nuvem

A infraestrutura em nuvem do PhaseGen é integralmente baseada no Supabase e deve ser configurada previamente à primeira execução do pipeline. Os seguintes componentes precisam estar implantados no projeto Supabase:

O schema do banco de dados com as cinco tabelas do modelo relacional, phases, enemy_types, sprites, background_layers e phase_enemies, incluindo todas as restrições de chave estrangeira com exclusão em cascata e as restrições CHECK nos campos de domínio controlado.

O bucket de armazenamento no Supabase Storage, configurado com política de acesso público à leitura, necessária para que o cliente de jogo possa baixar as texturas diretamente via HTTP sem autenticação.

As Edge Functions implantadas no projeto Supabase: a função phases, responsável por retornar a lista resumida de fases disponíveis, e a função phase, responsável por agregar e retornar todos os dados estruturais de uma fase específica em um único documento JSON.

6.1.4 Requisitos de Software — Cliente de Jogo

O cliente de jogo apresenta requisitos de implantação significativamente mais simples que o ambiente de geração. Em sua forma distribuída como executável compilado, requer apenas um sistema operacional Windows e conectividade de internet para acesso ao Supabase. Nenhum modelo de inteligência artificial, dependência Python ou instalação adicional é necessária no ambiente do jogador.

Para execução a partir do código-fonte, é necessária apenas a instalação da Godot Engine 4.5 e a abertura do projeto. A URL do Supabase e a chave pública (*anon key*) já estão embarcadas no código-fonte do autoload de comunicação com o Supabase, decisão deliberada, uma vez que exports da Godot Engine não possuem mecanismo nativo de variáveis de ambiente, e a anon key não carrega privilégios elevados, sendo restrita a operações de leitura por política de Row-Level Security no banco de dados. Nenhum asset visual está embutido no projeto: todos os recursos são obtidos dinamicamente em tempo de execução, de modo que o repositório do cliente de jogo não depende do repositório do backend para ser executado, desde que o projeto Supabase esteja corretamente configurado e a chave embarcada corresponda ao projeto correto.

As principais decisões de implantação refletem os princípios arquiteturais estabeleci-

dos no Capítulo 4: o desacoplamento entre geração e consumo permite que o cliente de jogo seja distribuído e executado de forma completamente independente do backend, enquanto a centralização dos assets no Supabase elimina a necessidade de qualquer sincronização de arquivos entre os ambientes. O único pré-requisito compartilhado entre os dois ambientes é o acesso ao mesmo projeto Supabase, que atua como ponto de integração entre o pipeline de geração e o cliente de jogo.

7 Manual do Usuário

Este manual descreve o uso do PhaseGen em seus dois contextos distintos: a operação do pipeline de geração de fases, destinada ao operador responsável pela execução do sistema; e a utilização do cliente de jogo, destinada ao jogador. Os procedimentos de instalação e configuração dos ambientes estão descritos no Capítulo 6.

7.1 Manual do Operador

Com o ambiente devidamente configurado conforme descrito no Capítulo 6, o dashboard de geração é iniciado executando o arquivo `start_server.bat` localizado na pasta de scripts. O script inicializa o servidor FastAPI e abre automaticamente o dashboard de monitoramento no navegador padrão, na porta 5000.

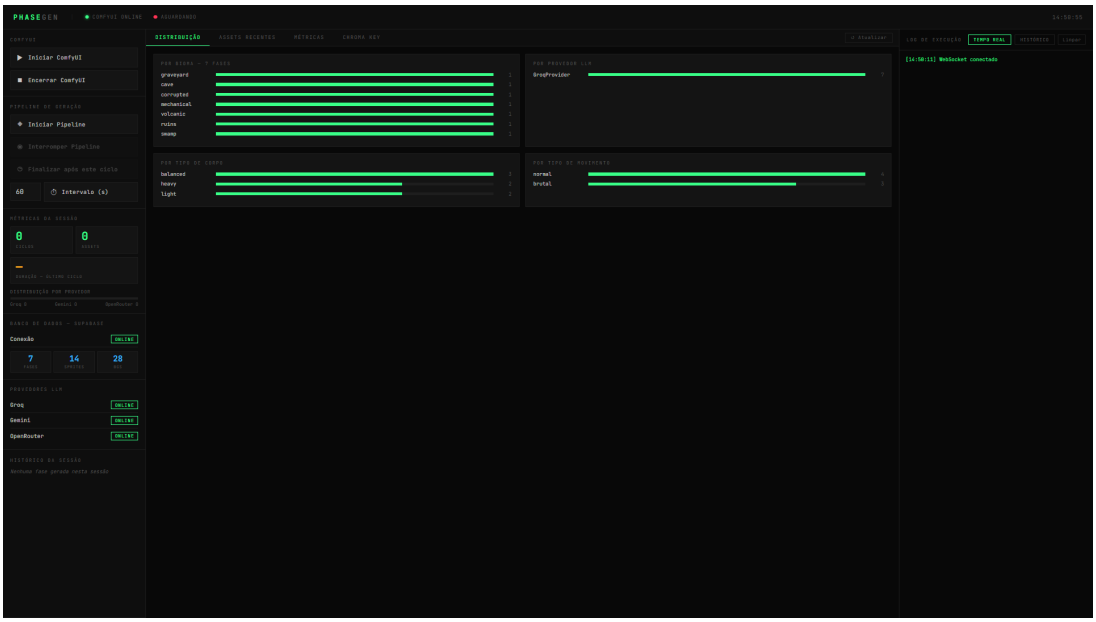

Figura 12: Dashboard de geração.

O painel de assets recentes apresenta as nove últimas imagens processadas pelo pipeline.

Figura 13: Assets recentes.

O painel de métricas, acessível no dashboard, exibe estatísticas acumuladas de todas as sessões de geração anteriores. As informações disponíveis incluem o número total de ciclos concluídos, o tempo médio por ciclo em segundos, a distribuição de fases geradas por tipo corporal de inimigo e por bioma, e a taxa de sucesso por provedor LLM.

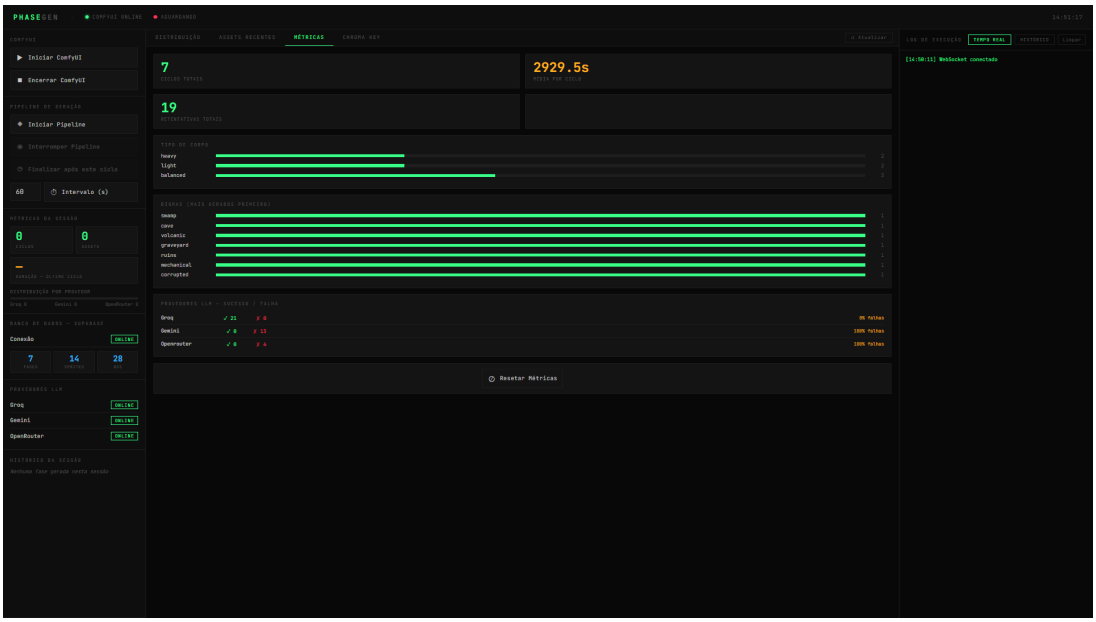

Figura 14: Métricas de geração.

O painel de configuração do *chroma key* permite ajustar os parâmetros de remoção de fundo dos assets gerados sem necessidade de reiniciar o servidor. Os parâmetros são organizados em uma grade com um campo de threshold por camada — *reference frame*, *spritesheet*, mid layer e intermediary layer — e campos opcionais de margem mínima e máxima de pixels removidos para ativação do ajuste automático de limiar.

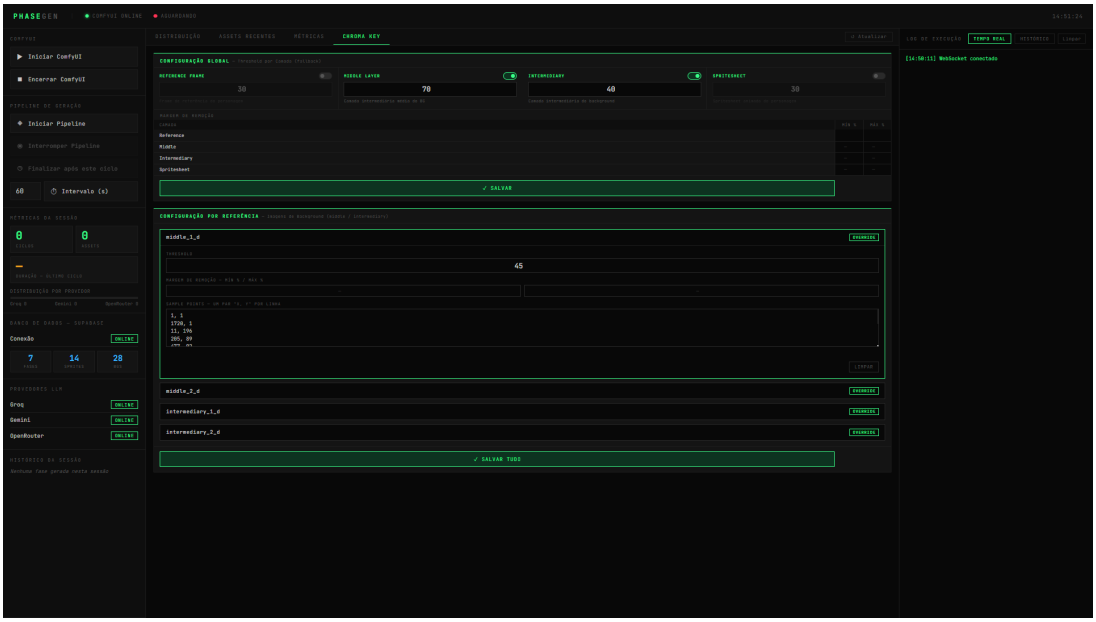

Figura 15: Configuração do *chroma key*.

Ao selecionar a opção "Histórico", no canto superior direito, é possível consultar os logs de geração das últimas sessões.

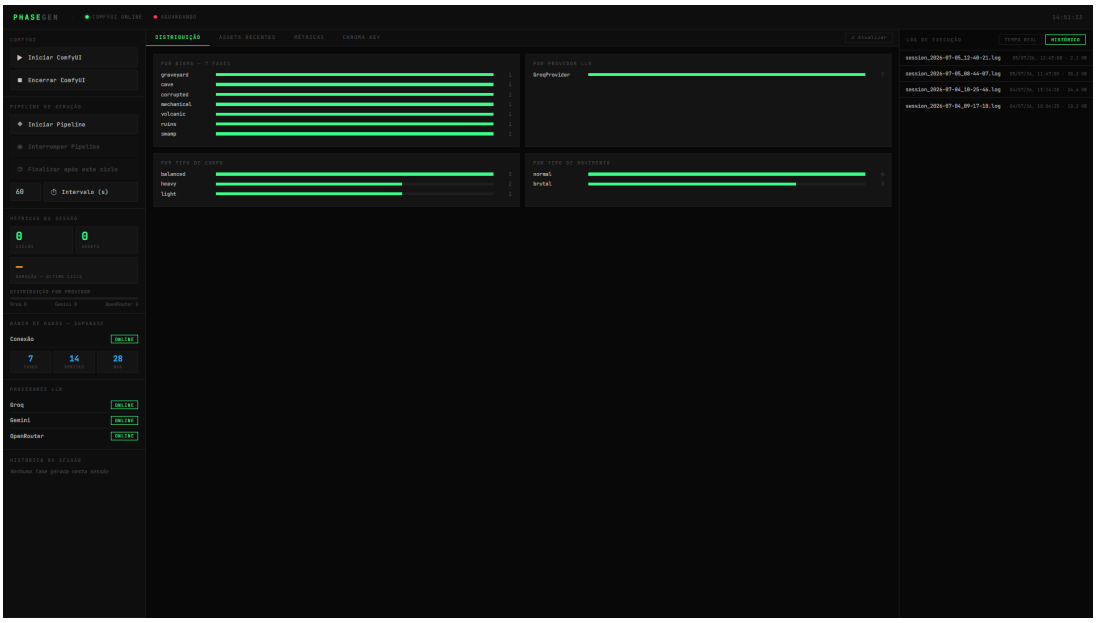

Figura 16: Histórico de sessões.

O pipeline pode ser iniciado clicando no botão **Iniciar Pipeline**. A partir desse momento, o sistema opera de forma autônoma, executando ciclos contínuos de geração sem necessidade de intervenção do operador. O banner de progresso exibido no topo do dashboard indica o bioma selecionado para o ciclo em andamento e a etapa atual de execução. O painel de log exibe em tempo real os eventos do pipeline — chamadas aos provedores LLM, submissão de jobs ao ComfyUI, aplicação do *chroma key* e persistência no Supabase.

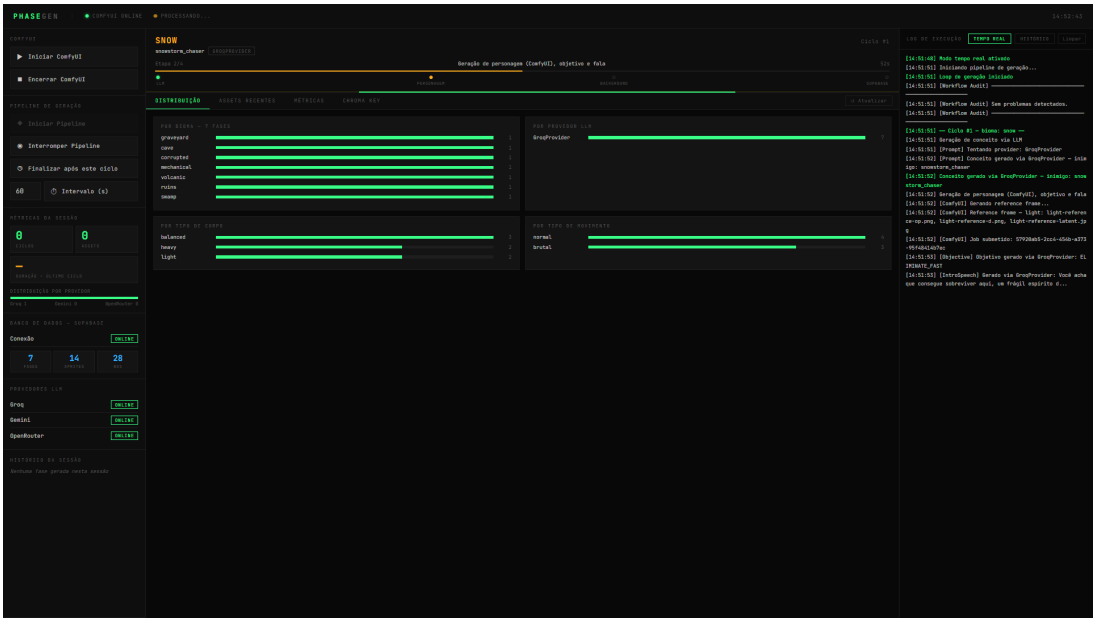

Figura 17: Ciclo de geração em andamento.

O operador dispõe de informações referentes às métricas daquela sessão, bem como à quantidade de assets salvos no banco de dados. São apresentadas também as seguintes opções na barra lateral esquerda:

- **Iniciar ComfyUI** — inicia o ComfyUI isoladamente, permitindo a edição e correção dos workflows.
- **Encerrar ComfyUI** — finaliza o processo do ComfyUI.
- **Iniciar Pipeline** — inicia todo o processo do pipeline de geração de fases.
- **Interromper Pipeline** — encerra o pipeline imediatamente, independentemente do estado do ciclo em andamento. Deve ser utilizado apenas em situações excepcionais, pois um rollback é executado e todo o progresso do ciclo em curso é perdido.
- **Finalizar após este ciclo** — sinaliza ao pipeline que deve encerrar ao término do ciclo em andamento, sem interromper a execução em curso. É o mecanismo recomendado para encerramento normal do sistema, pois garante a integridade de todos os assets gerados.
- **Intervalo** — ajusta o intervalo de tempo entre cada ciclo de geração.

7.2 Manual do Jogador

O PhaseGen é distribuído como executável no Windows. Para iniciar o jogo, execute o arquivo `PhaseGen.exe`. O jogo abre automaticamente em tela cheia e exibe uma tela de apresentação com informações sobre o projeto. A tela pode ser dispensada pressionando qualquer tecla ou clicando com o mouse.

Figura 18: Tela de Apresentação.

Após a tela de apresentação, o menu principal é exibido com as seguintes opções:

- **Jogar** — acessa a tela de seleção de fases.
- **Histórico** — exibe o registro de partidas anteriores.
- **Créditos** — exibe novamente a tela de apresentação do projeto.
- **Sair** — encerra o jogo.

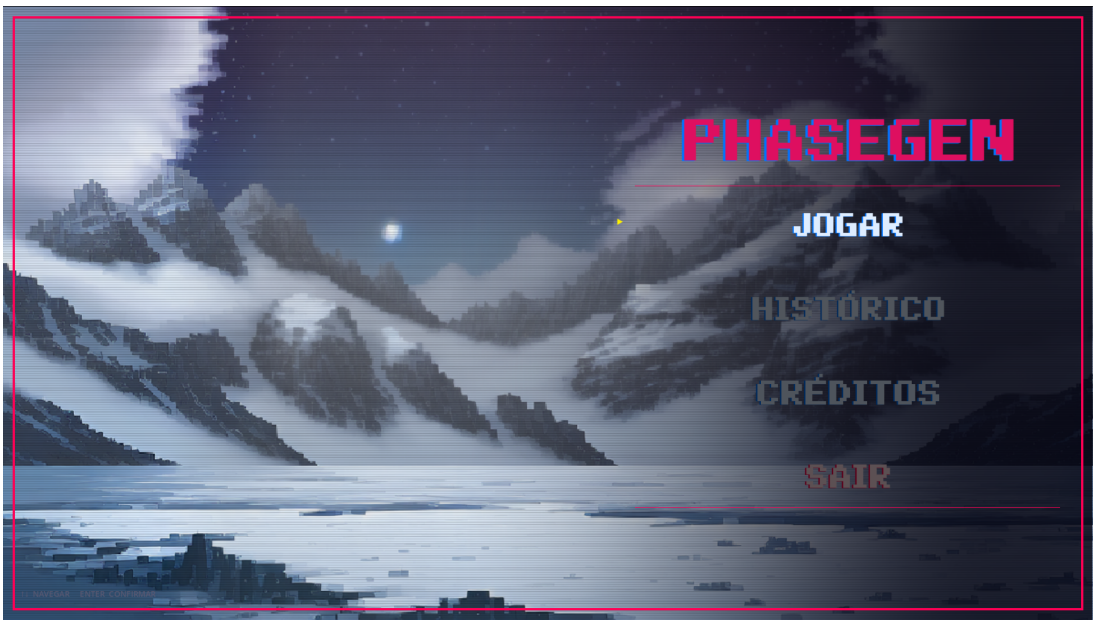

Figura 19: Menu inicial do jogo.

O histórico de partidas pode ser acessado pelo menu principal. A tela exibe todas as fases jogadas anteriormente, com o nome da fase, o resultado obtido (vitória ou derrota) — caso a fase tenha sido concluída — e a data em que foi jogada.

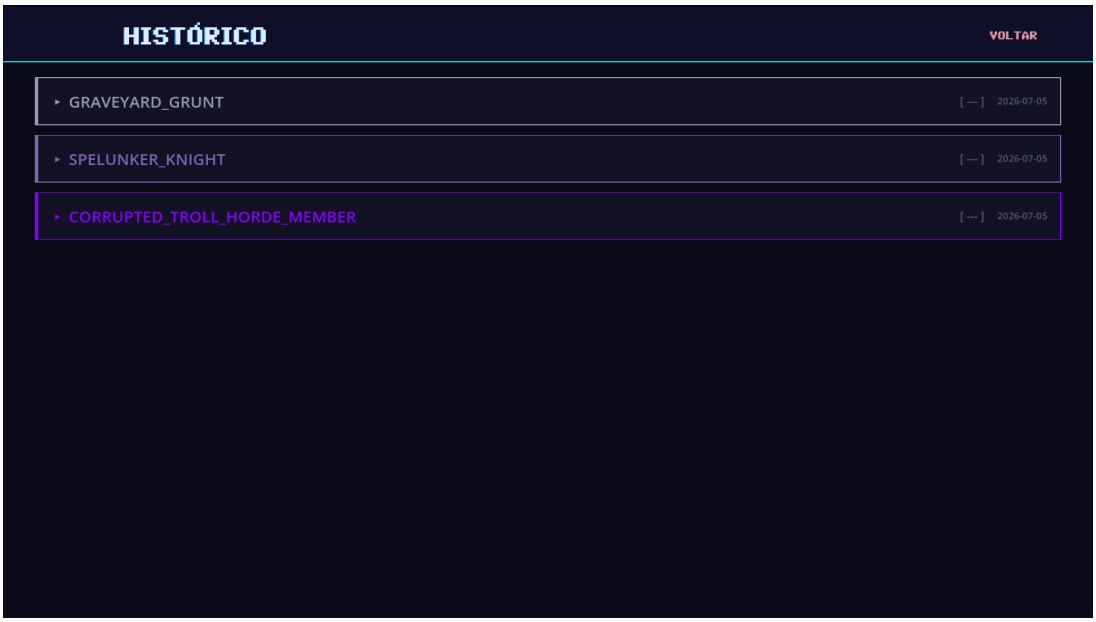

Figura 20: Tela de histórico de fases jogadas.

A tela de seleção de fases exibe as fases disponíveis em uma grade de seis itens por página, com controles de navegação entre páginas na parte inferior da tela. Cada card exibe a miniatura do cenário gerado, o nome da fase e um botão **Jogar**.

Figura 21: Menu de seleção de fase.

As miniaturas são carregadas progressivamente; enquanto o carregamento está em andamento, um indicador é exibido no lugar do grid. O botão **Jogar** de cada fase só é habilitado após o carregamento completo de sua miniatura. Ao clicar em **Jogar**,

uma tela com as informações de geração da fase é exibida antes do início da partida. A tela de informações exibe os prompts utilizados na geração dos assets, o provedor LLM responsável, o objetivo da fase e a fala de introdução do inimigo. Para iniciar a partida, clique em **Jogar**. Para retornar à seleção de fases, clique em **Voltar**.

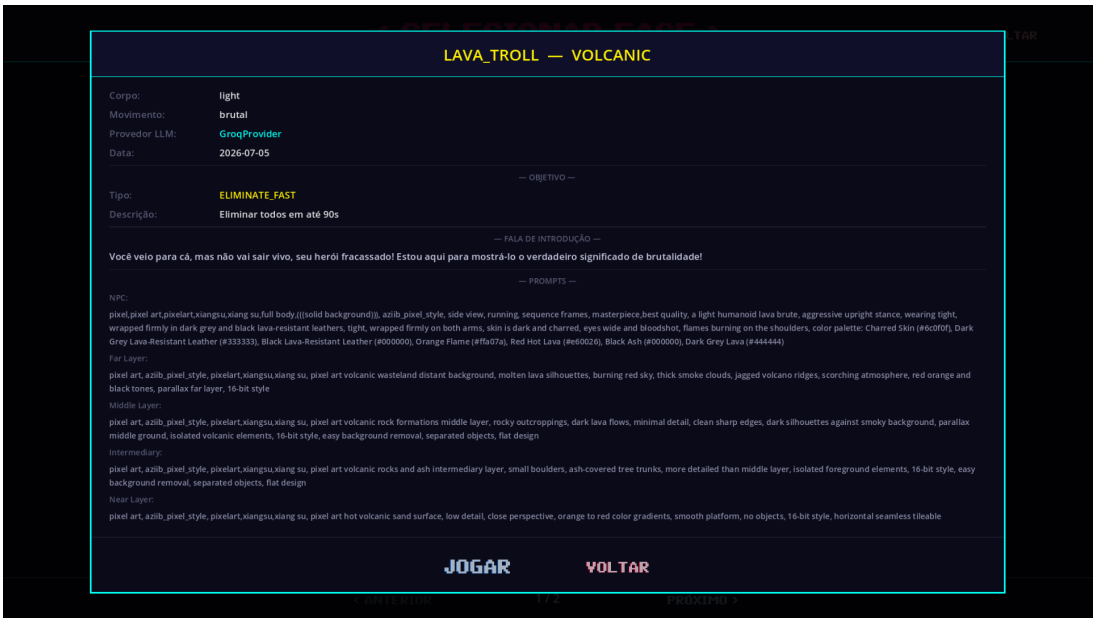

Figura 22: Informações de geração da fase escolhida.

Ao selecionar a fase, é solicitado que o jogador escolha o personagem. Entre as opções, é possível utilizar o personagem padrão, optar por uma escolha aleatória, ou selecionar o personagem de outra fase já gerada como personagem jogável.

Figura 23: Escolha de personagem.

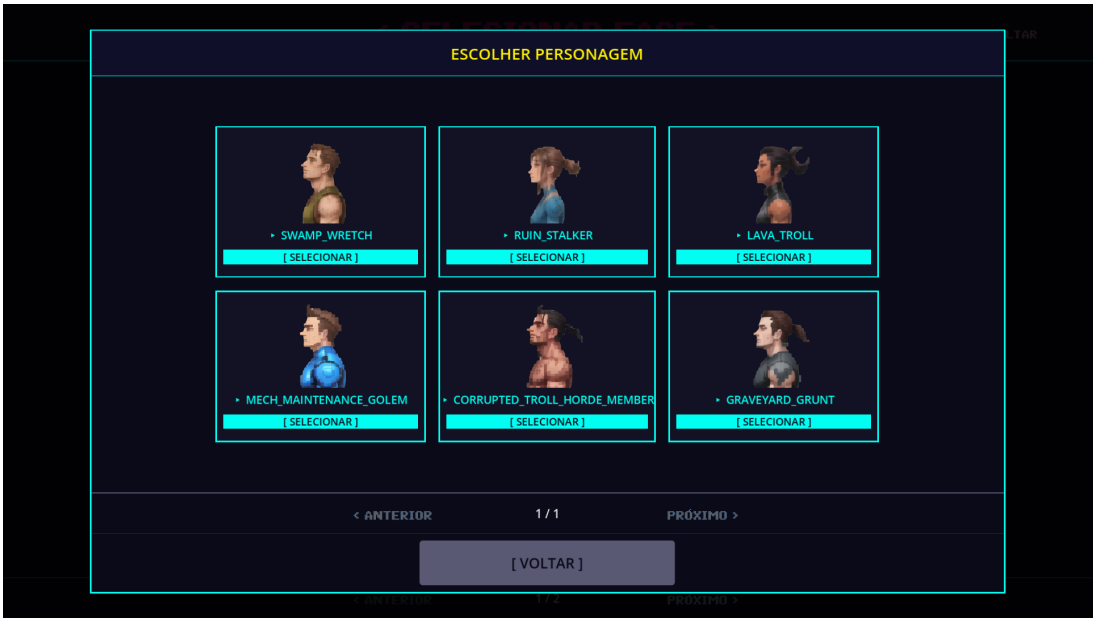

Figura 24: Escolher personagem de outra fase.

Ao iniciar a fase, uma tela de carregamento é exibida enquanto os assets são baixados do Supabase. Quando o carregamento é concluído, o inimigo apresenta-se ao jogador por meio de uma textbox com sua fala de introdução. A partida tem início após o jogador dispensar a textbox pressionando qualquer tecla ou clicando com o mouse.

Figura 25: Fala de introdução do inimigo.

Os controles do jogo são:

Tabela 8: Controles do jogo

Ação	**Tecla**
Mover para a esquerda	←
Mover para a direita	→
Pular	↑
Atacar	Espaço

O objetivo da fase é exibido no canto superior da tela, abaixo da barra de vida do jogador, junto ao indicador de progresso quando aplicável.

Figura 26: Visualização in-game.

A fase é concluída quando o objetivo é cumprido ou quando o jogador é derrotado. Em ambos os casos, uma tela de resultado é exibida com o desfecho da partida, o tempo decorrido e o dano total recebido pelo jogador.

Figura 27: Tela de conclusão da fase em caso de derrota.

Na tela de resultado, o jogador pode optar por jogar a fase novamente ou retornar ao menu principal.

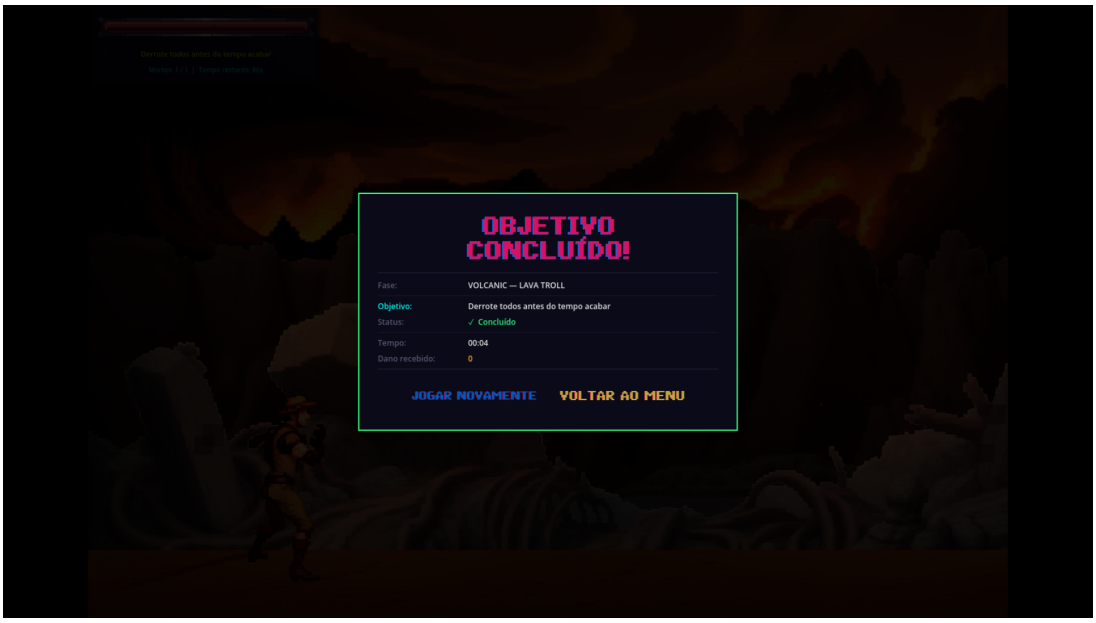

Figura 28: Tela de conclusão da fase em caso de vitória.

8 Conclusão

O presente trabalho investigou a viabilidade de integrar Modelos de Linguagem de Larga Escala e modelos generativos de imagem em um pipeline automatizado de geração procedural de assets visuais para jogos digitais, materializando essa investigação no desenvolvimento do PhaseGen. A proposta partiu da constatação de que a produção visual representa simultaneamente um dos maiores gargalos e um dos mais críticos fatores de qualidade no desenvolvimento independente de jogos, e buscou demonstrar, por meio de um sistema funcional completo, que essa integração constitui uma das estratégias possíveis, entre outras, para incorporar inteligência artificial generativa à rotina de desenvolvimento independente.

Os oito objetivos específicos estabelecidos foram integralmente alcançados. O pipeline de orquestração autônomo em Python coordena, sem intervenção humana, a comunicação entre os provedores LLM, o ambiente de geração de imagens ComfyUI, a infraestrutura de persistência Supabase e o motor de jogo Godot Engine 4.5. O sistema de geração de conceito via LLM produz, em uma única requisição estruturada, a temática do inimigo e os prompts necessários para todos os assets visuais da fase, garantindo coerência temática entre personagem e cenário. Os fluxos parametrizados no ComfyUI, condicionados por ControlNet e IP-Adapter, automatizam a produção de personagens e camadas de background em estilo pixel art com consistência estética entre os assets de um mesmo ciclo. Os métodos de remoção de fundo adaptados por tipo de asset, segmentação semântica via BRIA RMBG para o personagem e *chroma key* adaptativo para as camadas de cenário, preparam os assets gerados para uso com transparência no motor de jogo. A infraestrutura Supabase viabiliza o carregamento e a montagem dinâmica das fases em tempo de execução, sem que nenhum asset esteja embutido no executável. O sistema de objetivos procedurais gerados por LLM introduz variabilidade na dimensão do gameplay, tornando cada fase única não apenas esteticamente, mas também em termos de desafio proposto ao jogador. O dashboard de monitoramento em tempo real oferece observabilidade e controle operacional sobre o ciclo autônomo de geração. Por fim, o jogo desenvolvido na Godot Engine 4.5 materializa a proposta como ambiente de demonstração, com mecânicas de combate, *parallax scrolling* em quatro camadas independentes, efeitos visuais atmosféricos por bioma e carregamento dinâmico de todos os assets em tempo de execução.

Os resultados obtidos demonstram que a integração entre modelos de linguagem e modelos de difusão em um pipeline automatizado constitui uma abordagem viável para a produção autônoma de assets visuais em contextos de desenvolvimento independente, inclusive sob condições de hardware deliberadamente adversas, conforme validado pelos testes conduzidos em GPU sem aceleração nativa. O PhaseGen evidencia que é possível percorrer o caminho da descrição textual à fase jogável sem intervenção artística manual, deslocando o investimento de mão de obra especializada para tempo de processamento em hardware amplamente disponível no mercado consumidor. Ao mesmo tempo, as limitações identificadas, detalhadas na Seção 5.3, a variabilidade ocasional na qualidade da remoção de fundo por *chroma key*, a possibilidade de incoerências semânticas na geração textual mesmo com validade estrutural preservada, e a sensibilidade do tempo de ciclo à configuração de hardware e à disponibilidade de aceleração nativa, delimitam com clareza o estágio atual da tecnologia e os pontos que mais se beneficiariam de investigação adicional.

Espera-se que o presente trabalho contribua para a ampliação das possibilidades metodológicas no desenvolvimento de jogos independentes, oferecendo uma referência

concreta e funcional, não uma solução definitiva, para desenvolvedores que busquem incorporar, total ou parcialmente, estratégias de geração procedural assistida por inteligência artificial em seus próprios projetos.

8.1 Trabalhos Futuros

A partir das limitações identificadas na Seção 5.3, apontam-se as seguintes direções para trabalhos futuros:

- Comparação quantitativa entre diferentes modelos de difusão (SDXL, Stable Diffusion 3 e Flux), avaliando o compromisso entre qualidade visual e custo computacional;
- Avaliação do pipeline por desenvolvedores e artistas de jogos por meio de estudos de usabilidade, reduzindo o viés de autoavaliação identificado neste trabalho;
- Inclusão de métricas objetivas de qualidade de imagem (FID, LPIPS, CLIP Score) na avaliação dos assets gerados;
- Extensão da geração procedural para animações complementares, efeitos sonoros e trilhas musicais;
- Investigação de suporte à geração de assets para jogos 3D;
- Ajuste fino (*fine-tuning*) de modelos de difusão para estilos artísticos específicos, reduzindo a dependência de LoRAs de terceiros;
- Arquitetura distribuída para geração paralela em múltiplas GPUs, reduzindo o tempo médio de ciclo;
- Estudo comparativo de produtividade entre o fluxo de produção artística tradicional e o pipeline PhaseGen, com medição direta de tempo e esforço.

Agradecimentos

Ao professor Antonio Carlos dos Santos Souza, meu orientador, por acompanhar este trabalho e contribuir com seu direcionamento ao longo do processo.

À coordenadora Flávia Maristela, por ser sempre tão atenciosa e carinhosa com todos nós ao longo do curso, tornando essa caminhada mais acolhedora.

Aos demais professores do curso, pelos ensinamentos compartilhados ao longo desses anos, que foram essenciais para minha formação e para que eu chegasse até aqui.

Aos amigos que caminharam comigo nessa jornada, pela parceria de todos os dias, pelas risadas e pelo apoio que tornaram esses anos muito mais leves e memoráveis.

Referências

1. IGDA — INTERNATIONAL GAME DEVELOPERS ASSOCIATION. **Developer Satisfaction Survey 2023**. Western University, 2023. Acesso em: 5 julho 2026. Disponível em: `https://igda.org/dss`.

2. BADONI, Pankaj; KATAL, Avita; REDDY, M. Sweety; BHARGAVA, Mudit. **Graphics vs Gameplay: A Comparative Analysis in Gaming**. 2022. Acesso em: 3 maio 2026. Disponível em: `https://www.researchgate.net/publication/362940829_Graphics_vs_Gameplay_A_Comparative_Analysis_in_Gaming`.

3. TERNAR, Alexandru et al. **Generative AI in Game Development: A Qualitative Research Synthesis**. arXiv:2509.11898v2, Aalto University / University of York / University of Coimbra, dez. 2025. Acesso em: 5 julho 2026. Disponível em: `https://arxiv.org/abs/2509.11898`.

4. PYTHON SOFTWARE FOUNDATION. **Python 3 Documentation**. 2026. Acesso em: 3 maio 2026. Disponível em: `https://docs.python.org/3/`.

5. GROQ. **Llama 3.1 8B Instant — Model Documentation**. 2026. Acesso em: 3 maio 2026. Disponível em: `https://console.groq.com/docs/model/llama-3.1-8b-instant`.

6. GOOGLE. **Gemini API Documentation**. 2026. Acesso em: 3 maio 2026. Disponível em: `https://ai.google.dev/gemini-api/docs?hl=pt-br`.

7. OpenRouter, Inc. **OpenRouter Documentation**. 2026. Acesso em: 26 junho 2026. Disponível em: `https://openrouter.ai/docs/`.

8. ROMBACH, Robin; BLATTMANN, Andreas; LORENZ, Dominik; ESSER, Patrick; OMMER, Björn. **High-Resolution Image Synthesis with Latent Diffusion Models**. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), 2022, p. 10684–10695. Acesso em: 7 julho 2026. Disponível em: `https://arxiv.org/abs/2112.10752`.

9. THUNDER COMPUTE. **How to Run Stable Diffusion**. 2025. Acesso em: 4 julho 2026. Disponível em: `https://www.thundercompute.com/blog/how-to-run-stable-diffusion`.

10. ROMBACH, Robin; ESSER, Patrick. **Stable Diffusion v1-5 — Model Card**. Hugging Face, 2022. Acesso em: 7 julho 2026. Disponível em: `https://huggingface.co/stable-diffusion-v1-5/stable-diffusion-v1-5`.

11. PODELL, Dustin; ENGLISH, Zion; LACEY, Kyle; BLATTMANN, Andreas; DOCKHORN, Tim; MÜLLER, Jonas; PENNA, Joe; ROMBACH, Robin. **SDXL: Improving Latent Diffusion Models for High-Resolution Image Synthesis**. arXiv:2307.01952, 2023. Acesso em: 7 julho 2026. Disponível em: `https://arxiv.org/abs/2307.01952`.

12. CIVITAI. **ReV Animated**. 2023. Acesso em: 4 julho 2026. Disponível em: `https://civitai.com/models/7371/rev-animated`.

13. CIVITAI. **aziibpixelmix**. 2023. Acesso em: 4 julho 2026. Disponível em: https://civitai.com/models/195730/aziibpixelmix.

14. CIVITAI. **aziib pixel style**. 2024. Acesso em: 4 julho 2026. Disponível em: https://civitai.com/models/672328/aziib-pixel-style.

15. HUGGING FACE. **OrangeMixs VAE**. 2023. Acesso em: 4 julho 2026. Disponível em: https://huggingface.co/WarriorMama777/OrangeMixs/tree/main.

16. COMFY. **ComfyUI Documentation**. 2026. Acesso em: 3 maio 2026. Disponível em: https://docs.comfy.org/.

17. ZHO-ZHO-ZHO. **ComfyUI-BRIA_AI-RMBG**. GitHub, 2024. Acesso em: 4 julho 2026. Disponível em: https://github.com/ZHO-ZHO-ZHO/ComfyUI-BRIA_AI-RMBG.

18. ZHANG, Lvmin; RAO, Anyi; AGRAWALA, Maneesh. **Adding Conditional Control to Text-to-Image Diffusion Models**. In: Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV), 2023, p. 3836–3847. Acesso em: 7 julho 2026. Disponível em: https://arxiv.org/abs/2302.05543.

19. YE, Hu; ZHANG, Jun; LIU, Sibo; HAN, Xiao; YANG, Wei. **IP-Adapter: Text Compatible Image Prompt Adapter for Text-to-Image Diffusion Models**. arXiv:2308.06721, Tencent AI Lab, 2023. Acesso em: 7 julho 2026. Disponível em: https://arxiv.org/abs/2308.06721.

20. SUPABASE. **Supabase Documentation**. 2026. Acesso em: 3 maio 2026. Disponível em: https://supabase.com/docs.

21. CAO, Zhe; HIDALGO, Gines; SIMON, Tomas; WEI, Shih-En; SHEIKH, Yaser. **OpenPose: Realtime Multi-Person 2D Pose Estimation Using Part Affinity Fields**. IEEE Transactions on Pattern Analysis and Machine Intelligence, v. 43, n. 1, p. 172–186, 2021. arXiv:1812.08008. Acesso em: 7 julho 2026. Disponível em: https://arxiv.org/abs/1812.08008.

22. GODOT ENGINE. **Godot Engine Documentation**. 2026. Acesso em: 3 maio 2026. Disponível em: https://docs.godotengine.org/en/stable/.

23. HOLFELD, Julian. **On the relevance of the Godot Engine in the indie game development industry**. University of Kassel, 2024. Acesso em: 4 julho 2026. Disponível em: https://arxiv.org/abs/2401.01909.

24. GITHUB. **About GitHub and Git**. 2026. Acesso em: 3 maio 2026. Disponível em: https://docs.github.com/pt/get-started/start-your-journey/about-github-and-git.

25. PLANTUML. **PlantUML Language Reference Guide**. 2026. Acesso em: 3 maio 2026. Disponível em: https://plantuml.com/.

26. MAO, Xinyu; YU, Wanli; YAMADA, Kazunori D.; ZIELEWSKI, Michael R. **Procedural Content Generation via Generative Artificial Intelligence**. 2024. Acesso em: 3 maio 2026. Disponível em: https://arxiv.org/abs/2407.09013.

27. FARROKHI MALEKI, Mahdi; ZHAO, Richard. **Procedural Content Generation in Games: A Survey with Insights on Emerging LLM Integration**. 2024. Acesso em: 3 maio 2026. Disponível em: https://arxiv.org/abs/2410.15644.

28. FUKAYA, Kaisei; DAYLAMANI-ZAD, Damon; AGIUS, Harry. **Intelligent Generation of Graphical Game Assets: A Conceptual Framework and Systematic Review of the State of the Art**. 2023. Acesso em: 3 maio 2026. Disponível em: https://arxiv.org/abs/2311.10129.

29. WANG, Guanghui et al. **STED and Consistency Scoring: A Framework for Evaluating LLM Structured Output Reliability**. AWS Generative AI Innovation Center, 2025. Acesso em: 4 julho 2026. Disponível em: https://arxiv.org/abs/2512.23712.

30. LARABEL, Michael. **AMD ROCm 7.2 Now Released With More Radeon Graphics Cards Supported, ROCm Optiq Introduced**. Phoronix, 21 jan. 2026. Acesso em: 5 julho 2026. Disponível em: https://www.phoronix.com/news/AMD-ROCm-7.2-Released.

31. AMD. **ROCm 7.2.0 Release Notes**. ROCm Documentation, 2026. Acesso em: 5 julho 2026. Disponível em: https://rocm.docs.amd.com/en/docs-7.2.0/about/release-notes.html.

32. PUGET SYSTEMS. **Stable Diffusion Linux vs. Windows**. 2024. Acesso em: 4 julho 2026. Disponível em: https://www.pugetsystems.com/labs/articles/stable-diffusion-linux-vs-windows/.

A Workflow de geração do *spritesheet*

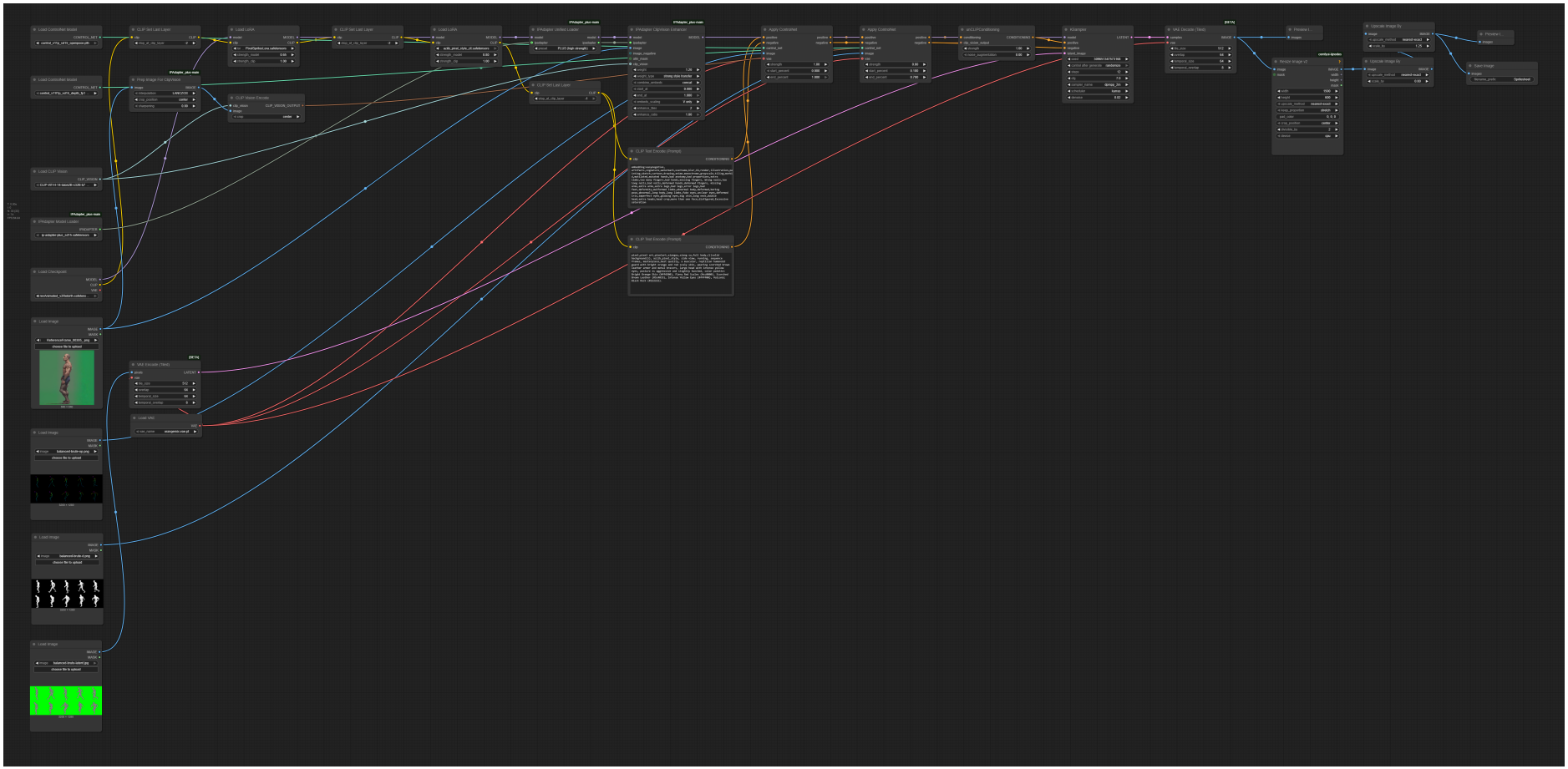

Figura 29: Workflow de geração do *spritesheet*

B Diagramas de Classes

B.1 Backend Python

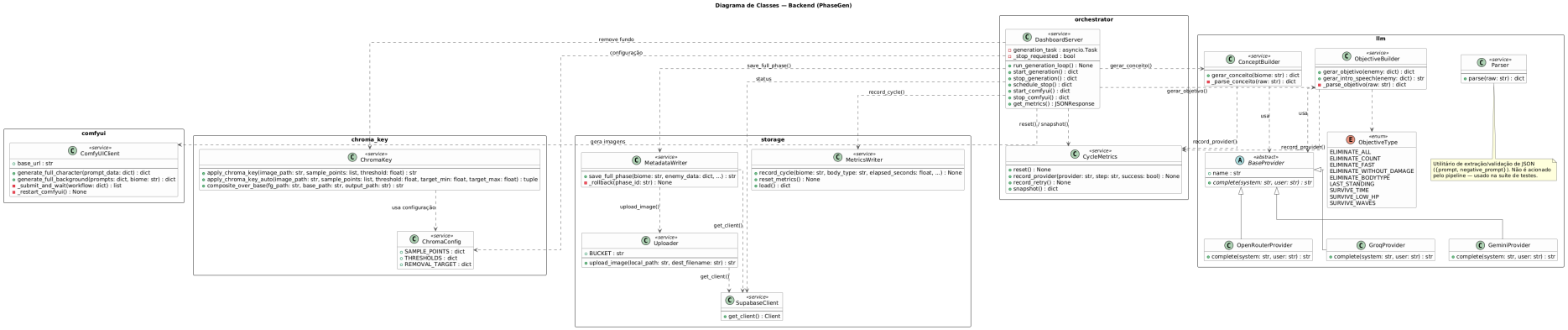

Figura 30: Diagrama de Classes — Backend Python

B.2 Cliente Godot — Autoloads e Interface

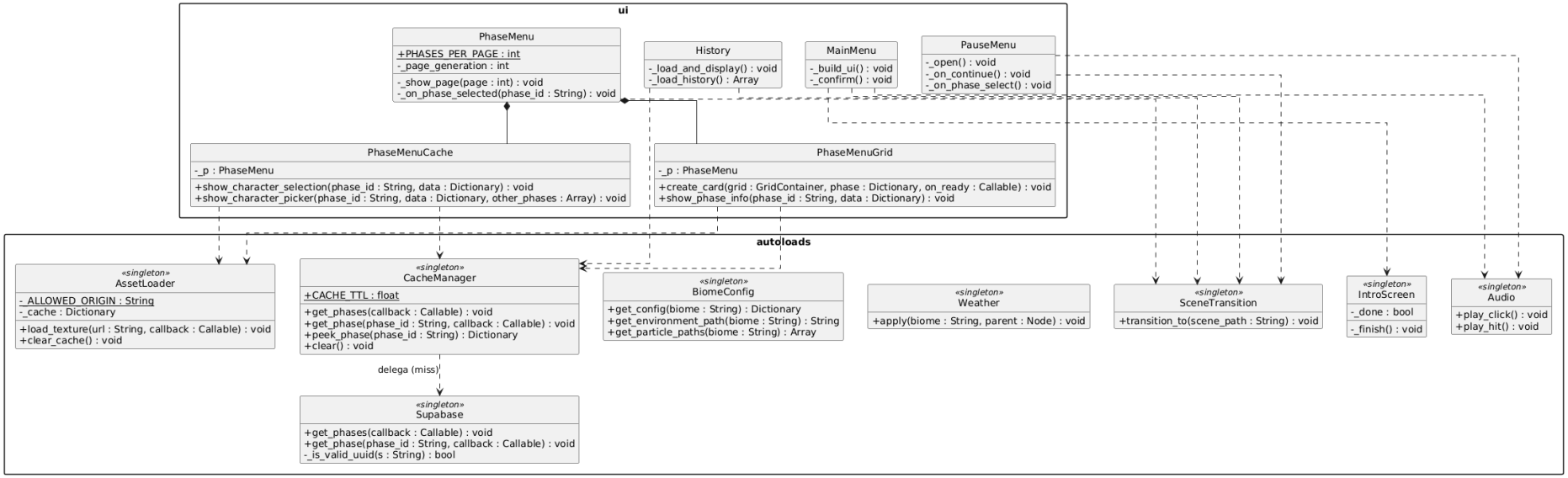

Figura 31: Diagrama de Classes — Cliente Godot: Autoloads e Interface

B.3 Cliente Godot — Phase e Entities

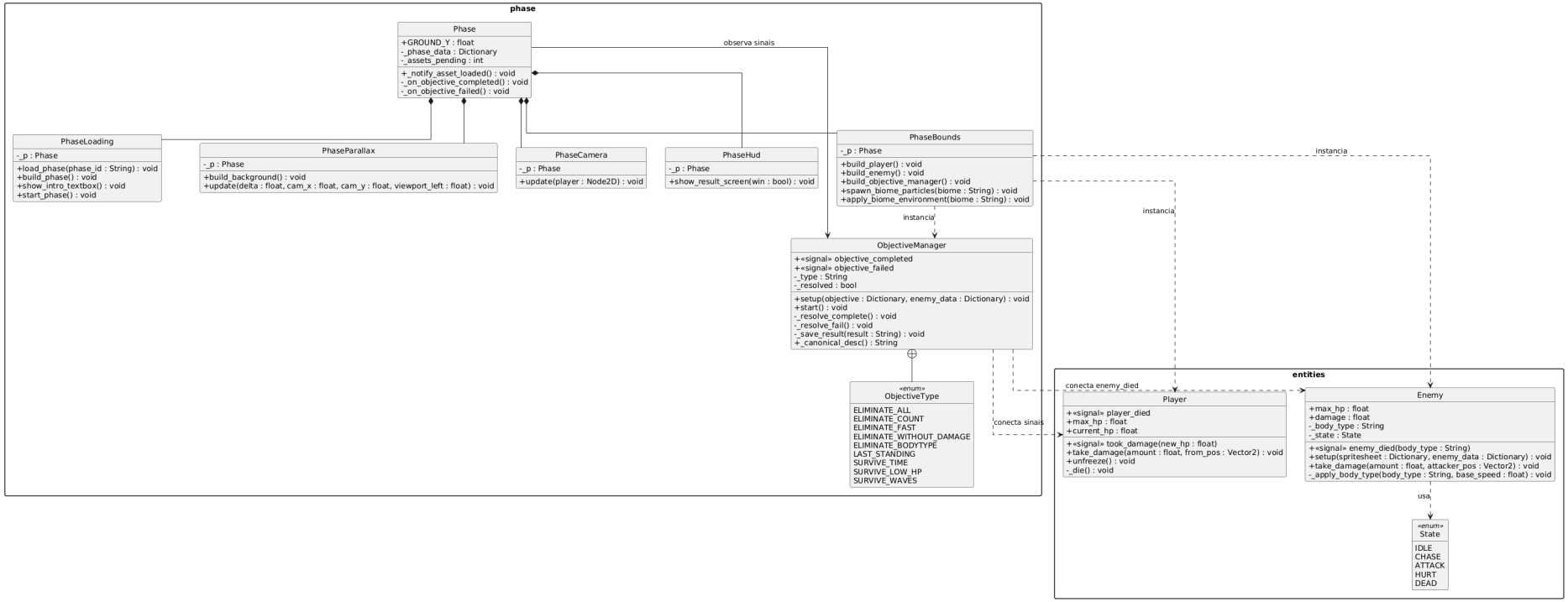

Figura 32: Diagrama de Classes — Cliente Godot: Phase e Entities