Constraint Bootstrapping

Technical report detailing the architecture of our Constraint Bootstrapping Proposal

James Vovos
Supervisors: Clinton Woodward and Charlotte Pierce

A technical report presented for the degree of
Bachelor of Computer Science

Faculty of Science, Computing and Engineering Technologies
Swinburne University of Technology
Australia
17/05/2023

List of Figures

Contents

1 Introduction

1.1 Overview

This technical report aims to cover and discuss the architecture and development process involved in creating our constraint bootstrapping architecture. The architecture aims to introduce an alternative method to generating 2D game assets, by utilising deep learning artificial intelligence models.

1.2 Application Context

Our domain specific context primarily focuses on generating 2D game assets involved in creating non-player character quests; such as items, storyline, dialogue, objectives and non-player character backstories. However, our architecture can easily be extended to create more technical assets, due to its modular structure and fine-tuned text-to-image generation models. The attached research report can also provide more context, research outcomes and results to supplement this paper.

2 Tools and Frameworks

2.1 OpenAI GPT-3.5

Our architecture leveraged OpenAI's, GPT-3.5 large language model (LLM) to process and generate text output for our project. This was to utilise it's highly capable neural network trained with approximately 570GB of text data, and over 175 billion parameters. ChatGPT was used to process instructions and deliver output based on the hyper-training our architecture provided it.

2.2 Stable Diffusion and DreamBooth

Stable Diffusion was used to generate 2D game asset renders. Stable Diffusion is an open-source, deep learning, text-to-image model. The model itself produced low-quality outputs on it's standard training data- However, we were able to take advantage of new research provided by Google, utilising a technology they created called "DreamBooth" to fine-tune and refine the output of Stable Diffusion's text-to-image results in a style that matched our preferences. This was done through our own means of training data. See Figure 1 for more details on Stable Diffusion's base model vs our fine-tuned models outputs using the same text-to-image prompt.

(a) Stable Diffusion Base Model

(b) Fine Tuned Stable Diffusion Model

Figure 1: Stable Diffusion Base Model vs Fine-Tuned Model Output, with Prompt: magical potion, blue, rpg based game, 8K octane render, photorealistic.

2.3 Leonardo AI

Leonardo AI is a fine-tuned text-to-image generation model that has been trained on top of the base Stable Diffusion model. The API and web interface were utilised to custom train our own image generation models using curated data sets. Behind the scenes, the training utilises DreamBooth, which allows us to train our model without the need to utilise GPU and CPU intensive resources on our local machine. Alternatively, we could utilise Google Colab to create a fine-tuned model by following examples from ShivamShrirao (2023) GitHub repository.

2.4 LangChain

LangChain is a Python framework that facilitates the integration of different components to enhance outputs based on LLMs. Specifically, it addresses limitations in LLMs like OpenAI's GPT-3.5 model, where it can only generate a certain amount of tokens for a given response. It also lacks memory capabilities. For example: If you enter in a question, the model will generate a response and forget the context of that answer when answering the next question. This is especially true when a conversation with the model exceeds certain token limits.

To overcome these constraints, we leveraged LangChain's chaining library to seamlessly integrate this capability into our architecture. For instance, suppose we aimed to generate a quest for a non-player-character based on a character description. While ChatGPT can handle the initial request, generating character dialogue or quest objectives becomes a manual process. However, with LangChain, we can pass earlier outputs as input parameters to subsequent functions or chains, establishing an architecture pipeline to automate and streamline these tasks.

2.5 Streamlit

Streamlit is an open-source framework for Machine Learning and Data Science. It allowed us to create a web app to host our architecture playground using Python.

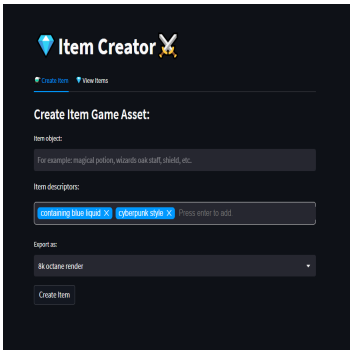
3 Interface Design

3.1 Creating the User Interface

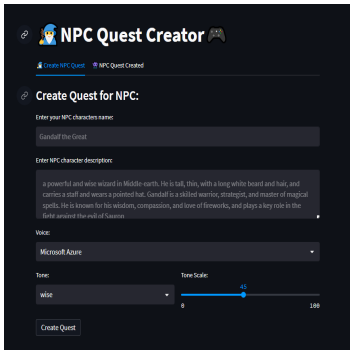
We created a basic web app in Python utilising the Streamlit library. This allowed us to showcase the asset creation pipeline for generating multiple 2D game assets. The user-interface aims to showcase the ease of use and efficiency in generating 2D game assets, from minimal human input.

2D game assets such as game world items, textures, non-player character quests, objectives, items and even character backstories and dialogue can all be generated from a simple prompt.

Figure 2 illustrates the interface design for some of the 2D game assets the application allows you to create. These can all be customised and tailored to generate outputs according to a user's stylistic preferences. More details on modifying the stylistic preferences can be found within the Building the Game Asset Creation Pipeline section of this report.

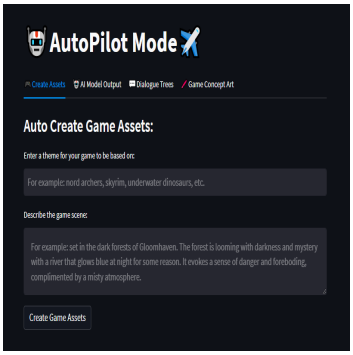


(a) Item Creator

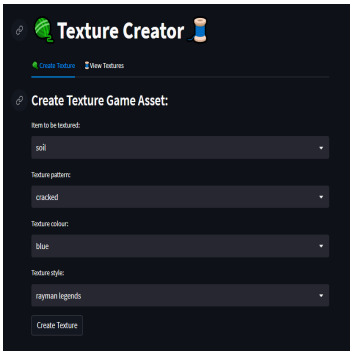


(b) NPC Quest Creator

(c) Isometric Worlds Creator



(d) Autopilot Mode



(e) Texture Creator

Figure 2: Web Interface Design Examples

4 Architecture Concept Overview

4.1 Initial Thoughts and Hypothesis

The initial idea or concept behind our architecture was to utilise large language models (LLMs) to aid in building text-to-image generation prompts. LLMs were found to be exceedingly capable of generating high quality descriptive summary texts and descriptions. These descriptions were leveraged, so that they could then be passed to text-to-image generation models, to render high quality game asset items based on these descriptors.

We quickly discovered that the use case for generating high quality text output could be tweaked to generate other game assets that didn't require image rendering - such as quest storyline or dialogue.

Based on our initial concept we hypothesised that we could:

- Utilise GPT-3.5 to generate text and item descriptors to pass to our text-to-image models.
- Utilise GPT-3.5 to create text based game assets such as quest storyline, dialogue, non-player character descriptions, etc.

4.2 The Problem with our Initial Hypothesis

The problem or constraint with our initial hypothesis arises when we attempt to generate multiple game assets simultaneously. Unfortunately, OpenAI's GPT-3.5 model constantly forgets the context of the previous prompt you provide it based on it's own token limit architecture. This means if we wanted to create multiple game assets we would manually need to setup the prompts for each game asset.

For example: If we wanted to generate 2D game assets for a quest (such as quest storyline, objectives, dialogue, non-player character backstories and items), we would need to create the storyline with a prompt first, iterate on the prompt, then proceed to create the quest objectives, iterate on the prompt, etc.

The prompt memory allocation limit (what we frame as memory tokenization limits), posed as a constraint to our initial concept or idea that we had to overcome in order to automate the asset creation pipeline. The next section in this report demonstrates how we overcame this constraint using our constraint bootstrapping architecture.

5 Building the Game Asset Creation Pipeline

5.1 LangChain Templates

In order to generate 2D game assets, we needed a way to tell ChatGPT's large language model (LLM) how to generate the assets we required. This is done through the use of "prompts" in which we create using LangChain's PromptTemplate objects.

Essentially, we create a template which takes sample inputs, and generates an output based on the custom prompt we created.

As an example, Figure 3 highlights how we provide the initial game theme and details as input, and based on this data, return a quest storyline response as output from the model.

```
from langchain.prompts import PromptTemplate

"""Prompt Templates"""

# NOTE: Chain 1: Sample inputs: {theme: "skyrim"}, {details: "set in the dark forests of Gloomhaven.
The forest is looming with darkness and mystery with a river that glows blue at night for some reason.
It evokes a sense of danger and foreboding, complimented by a misty atmosphere."}

# Create game storyline based on the initial concept theme and game scene details.
# Inputs: {"theme", "details"}
# Outputs: {"storyline"}
storyline_template = PromptTemplate(
    input_variables=["theme", "details"],
    template="create a short storyline for an rpg based game based around this concept theme: {theme}.
Then modify the storyline based on these provided additional details regarding the game scene:
{details} IMPORTANT: DO NOT include the Original storyline in your response, only provide the modified
storyline.",
)
```

Figure 3: LangChain Templates Create Quest Storyline

Our next template required that we utilise the output generated from the previously generated template, and return two non-player characters for our quest. This allows us to overcome ChatGPT's memory allocation or tokenization limit barrier we discussed earlier within this report. Essentially, we pass the output generated as input to our next chain in our asset creation pipeline. Concepts and ideas regarding how to integrate this component into our architecture were drawn from, and inspired by Park et al. (2023).

Figure 4 illustrates how the storyline output generated in Figure 3 is chained to the next stage to generate non-player quest characters. A full list of the templates used can be viewed within the appendix section of this report.

```
# Create 2 NPC Characters based on the game storyline.
# Inputs: {"storyline"}
# Outputs: {"characters"}
characters_template = PromptTemplate(
    input_variables=["storyline"],
    template="from the game storyline: {storyline}, create 2 fictional game characters (NPCs). List
them and a brief character description (1-2 sentences each).",
)
```

Figure 4: LangChain Templates Create Quest Characters

5.2 Chaining Together the Templates

In order to chain the templates together, (providing automation in our game asset generation tool), we utilised LangChain's LLMChain objects. These objects make us of the templates created earlier as prompts and return the desired output we requested in our templates. The chains work by passing the output from our current chain to our next chain until all required outputs are delivered via the model sequentially and automatically.

Figure 5 demonstrates how the LLMChain objects utilise the templates as prompts, in order to chain together the outputs sequentially. The output is then passed to the next chain in the pipeline, providing automation in our asset generation pipeline.

```
#OpenAI
llm = ChatOpenAI(temperature=0.7, model_name="gpt-3.5-turbo")
# API model costs: https://openai.com/pricing

# NOTE: Chain 1

storyline_chain = LLMChain(
    llm=llm,
    prompt=storyline_template,
    verbose=True,
    output_key="storyline"
)

characters_chain = LLMChain(
    llm=llm,
    prompt=characters_template,
    verbose=True,
    output_key="characters"
)

# NOTE: Chain 2

backstories_chain = LLMChain(
    llm=llm,
    prompt=backstories_template,
    verbose=True,
    output_key="backstories"
)
```

Figure 5: LangChain Chains Objects Examples

5.3 Creating the Pipeline

Our goal was to ensure the structure of our architecture remained modular in nature. This would enable developers to modify the architecture and create different asset creation pipelines depending on the type of assets they would like to create.

To do this, we developed a Pipeline base class, which other asset creation pipelines can inherit from to be able to modify their generated output (as different asset types will generate different things). This is done by chaining together the stages of the pipeline for that asset type, which is then executed sequentially via the run() function.

Figure 6 illustrates the implementation of our Pipeline base class and child class inheritance structure.

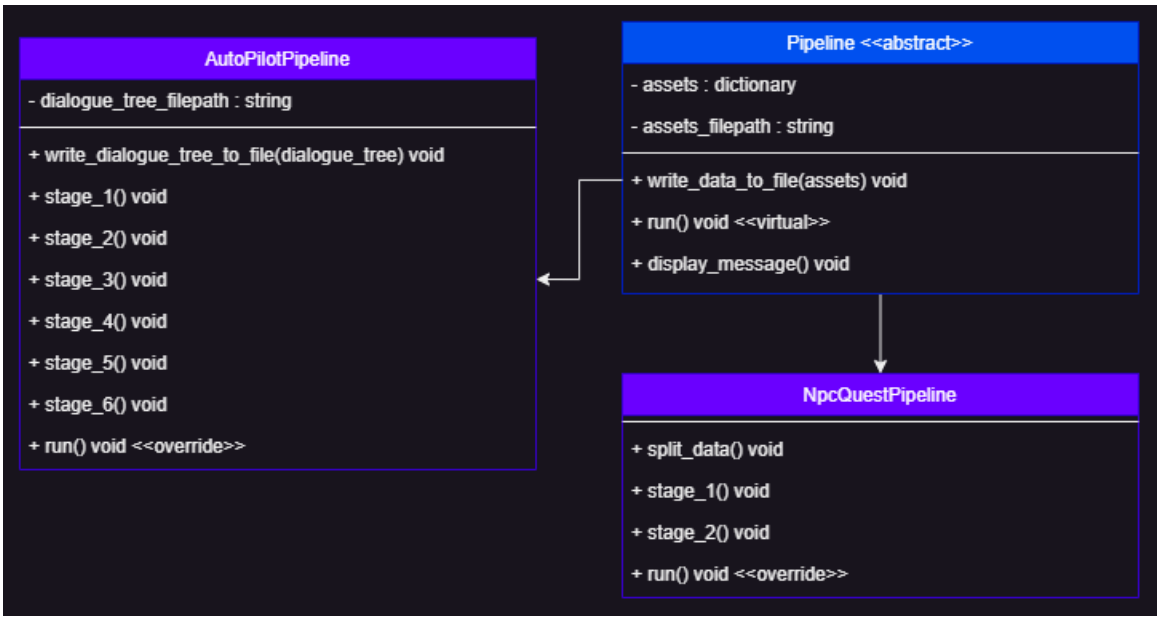


Figure 6: Pipeline Class UML Diagram

Figure 7 highlights the properties of the Pipeline base class. The child class is passed in assets and a file path to which to retrieve and save newly created assets to via JSON local storage. The run() function will be modified based on the pipeline staging processes required to generate the specific game asset.

```python
from abc import ABC
import json
import streamlit as st


""" Pipelines with stages to execute LangChain chaining events, based on the type of asset creation
method the user chooses (for example: autopilot mode, create items, create tilemaps, etc). """


# Abstract base class
class Pipeline(ABC):
    def __init__(self, assets: dict, assets_filepath: str):
        self.assets = assets
        self.assets_filepath = assets_filepath

    def write_data_to_file(self, assets):
        # Append data from pipeline to file.
        with open(self.assets_filepath, "w") as f:
            json.dump(assets, f, indent=4)
        f.close()

    def run(self):
        pass

    def display_message(self):
        # Display success message once completed.
        st.info('Game assets created successfully.')
        st.snow()
```

Figure 7: Pipeline Abstract Base Class

As an example, our pipeline for automatically generating quest related game assets is split into multiple stages or components. Figure 8 showcases the implementation of the AutoPilot Pipeline Class inheriting from the base class.

```python
from langchain.chains import SequentialChain
from components.chains.autopilot.chains import storyline_chain, characters_chain, backstories_chain,
quests_chain, dialogue_chain, items_chain, item_descriptions_chain, dialogue_tree_chain
from components.pipelines.pipeline import Pipeline
import json


""" Creates an AutoPilot pipeline with it's own chaining stages """

class AutoPilotPipeline(Pipeline):
    def __init__(self, assets: dict, assets_filepath: str, dialog_tree_filepath: str):
        super().__init__(assets, assets_filepath)
        self.assets = assets
        self.assets_filepath = assets_filepath
        self.dialogue_tree_filepath = dialog_tree_filepath

    # Write the branching-dialogue into seperate JSON file to display tree.
    def write_dialogue_tree_to_file(self, dialogue_tree):
        with open(self.dialogue_tree_filepath, 'w') as f:
            json.dump(dialogue_tree, f, indent=4)
        f.close()
```

Figure 8: AutoPilot Pipeline Class

In order for us to automatically create game assets such as items, storyline, dialogue and character backstories, we can pass in previously generated output as input to our next pipeline stage, via LangChain's chaining functionality.

For example, in Stage 1, we chain together the chains we created earlier utilising LangChain's sequential chain object, and add these newly created assets to a JSON file locally. You can see moving from stage 1 to stage 2 that we create a new chain - where we essentially, pass the storyline and characters created from Stage 1 of our pipeline process and generate character backstories from it. We utilise the previously generated output, and leverage this as our input in our next chain.

Figure 9 and 10 illustrate the multiple pipeline stages or components involved in generating quest related game assets via the AutoPilot Pipeline.

```python
    # Chaining pipeline stage 1
    def stage_1(self):
        # Get required input data from JSON file
        theme = self.assets['theme']
        details = self.assets['details']

        # Chain the data together in sequential order
        chain_1 = SequentialChain(chains=[storyline_chain, characters_chain], verbose=True,
input_variables=["theme", "details"], output_variables=["storyline", "characters"])

        # Get AI model output response
        output = chain_1({'theme': theme, 'details': details})

        # Grab the values from the first chain that ran
        storyline = output['storyline']
        characters = output['characters']

        # Add assets to JSON file to be retrieved by other chains
        self.assets['storyline'] = storyline
        self.assets['characters'] = characters

        # Write the updated data back to the assets file
        self.write_data_to_file(self.assets)
```

(a) Stage 1

```python
# Chaining pipeline stage 2
def stage_2(self):
    # Get required input data from JSON file
    storyline = self.assets['storyline']
    characters = self.assets['characters']

    # Chain the data together in sequential order
    chain_2 = SequentialChain(chains=[backstories_chain], verbose=True, input_variables=["storyline",
"characters"], output_variables=["backstories"])

    # Get AI model output response
    output = chain_2({'storyline': storyline, 'characters': characters})

    # Add assets to JSON file to be retrieved by other chains
    self.assets['backstories'] = output['backstories']

    # Write the updated data back to the assets file
    self.write_data_to_file(self.assets)
```

(b) Stage 2

```python
# Chaining pipeline stage 3
def stage_3(self):
    # Get required input data from JSON file
    backstories = self.assets['backstories']
    storyline = self.assets['storyline']

    # Chain the data together in sequential order
    chain_3 = SequentialChain(chains=[quests_chain], verbose=True, input_variables=["backstories",
"storyline"], output_variables=["quests"])

    # Get AI model output response
    output = chain_3({'backstories': backstories, 'storyline': storyline})

    # Add assets to JSON file to be retrieved by other chains
    self.assets['quests'] = output['quests']

    # Write the updated data back to the assets file
    self.write_data_to_file(self.assets)
```

(c) Stage 3

Figure 9: Pipeline Stages 1-3

```python
# Chaining pipeline stage 4
def stage_4(self):
    # Get required input data from JSON file
    quests = self.assets['quests']
    backstories = self.assets['backstories']

    # Chain the data together in sequential order
    chain_4 = SequentialChain(chains=[dialogue_chain], verbose=True, input_variables=["quests",
"backstories"], output_variables=["dialogue"])

    # Get AI model output response
    output = chain_4({'quests': quests, 'backstories': backstories})

    # Add assets to JSON file to be retrieved by other chains
    self.assets['dialogue'] = output['dialogue']

    # Write the updated data back to the assets file
    self.write_data_to_file(self.assets)
```

(a) Stage 4

```python
# Chaining pipeline stage 5
def stage_5(self):
    # Get required input data from JSON file
    dialogue = self.assets['dialogue']
    quests = self.assets['quests']

    # Chain the data together in sequential order
    chain_5 = SequentialChain(chains=[items_chain, item_descriptions_chain], verbose=True,
input_variables=["dialogue", "quests"], output_variables=["items", "item_descriptions"])

    # Get AI model output response
    output = chain_5({'dialogue': dialogue, 'quests': quests})

    # Add assets to JSON file to be retrieved by other chains
    self.assets['items'] = output['items']
    self.assets['item_descriptions'] = output['item_descriptions']

    # Write the updated data back to the assets file
    self.write_data_to_file(self.assets)
```

(b) Stage 5

```python
# Chaining pipeline stage 6
def stage_6(self):
    # Get required input data from JSON file
    dialogue = self.assets['dialogue']

    # Chain the data together in sequential order
    chain_6 = SequentialChain(chains=[dialogue_tree_chain], verbose=True, input_variables=["dialogue"],
output_variables=["dialogue_tree"])

    # Get AI model output response
    output = chain_6({'dialogue': dialogue})

    # Add assets to JSON file to be retrieved by other chains
    dialogue_tree = output['dialogue_tree']

    # Conver the dialogue text into JSON format for branching-dialogue tree view
    self.write_dialogue_tree_to_file(dialogue_tree)
```


(c) Stage 6

Figure 10: Pipeline Stages 4-6

5.4 Running the Pipeline

Figure 11 demonstrates the functionality of the run() function for the AutoPilot Pipeline. This ensures that the implementation for this specific asset creation pipeline (Generating Sample Game Quests), could be modified to to sequentially extract the asset generations automatically in the order that we expect or prefer.

```
# Execute the pipeline stages
def run(self):
    # Run Chain 1
    self.stage_1()
    # Run Chain 2
    self.stage_2()
    # Run Chain 3
    self.stage_3()
    # Run Chain 4
    self.stage_4()
    # Run Chain 5
    self.stage_5()
    # Run Chain 6
    self.stage_6()
    # Display message once complete
    self.display_message()
```

Figure 11: AutoPilot Pipeline Run() Function

5.5 Creating the Game Assets

Once our pipeline and sequential chaining process is setup, our game assets are ready to be created. All we have to do now is pass our model an initial prompt. In the case of our AutoPilot Pipeline, our initial prompt was an overall theme and description of the game theme or location.

Figure 12 highlights how we utilised the Streamlit library as our web interface, to gather input data on the client side seamlessly.

```
# Web App UI
st.title("👨‍✈️AutoPilot Mode✈️")
# Insert containers separated into tabs:
create_tab, data_tab, dialogue_tab, game_art_tab = st.tabs(["🎮Create Assets", "👨‍✈️AI Model Output", "💬
Dialogue Trees", "🖌️Game Concept Art"])
with create_tab:
    st.subheader("Auto Create Game Assets:")

    theme = st.text_input("Enter a theme for your game to be based on:", placeholder="For example: nord
archers, skyrim, underwater dinosaurs, etc. ")
    details = st.text_area("Describe the game scene:", placeholder="For example: set in the dark forests
of Gloomhaven. The forest is looming with darkness and mystery with a river that glows blue at night for
some reason. It evokes a sense of danger and foreboding, complimented by a misty atmosphere.")

    st.button(label="Create Game Assets", on_click=display_spinner)

with data_tab:
    st.subheader("Game Scene:")
    # load assets created to UI
    with st.expander("Storyline:"):
        st.info(assets['storyline'])

    with st.expander("Characters:"):
        st.info(assets['characters'])

    with st.expander("Backstories:"):
        st.info(assets['backstories'])

    with st.expander("Objectives:"):
        st.info(assets['quests'])

    with st.expander("Dialogue:"):
        st.info(assets['dialogue'])

    with st.expander("Items:"):
        st.info(assets['items'])

    with st.expander("Items with descriptions:"):
        st.info(assets['item_descriptions'])
```

Figure 12: Grabbing Initial Input Prompt

The data was then passed to our pipeline chaining process to generate game assets for game quests. Figure 13 showcases the creation and invocation of a sample asset creation pipeline.

```python
# Create game assets
def create_assets(theme: str, details: str,):
    if theme != "" and details != "":
        # Assign the data
        assets['theme'] = theme
        assets['details'] = details

        # Write data to file
        with open(assets_filepath, 'w') as f:
            json.dump(assets, f, indent=4)

        # Run the pipeline
        pipeline = AutoPilotPipeline(assets, assets_filepath, dialogue_tree_filepath)
        pipeline.run()

    else:
        st.error('Please fill out all provided fields.')
```

Figure 13: Create Assets Function

Figure 14 provides a more visual overview of the structure of the architecture, focusing on how basic inputs are turned into more advanced outputs via custom pipelines. It showcases how a pipeline can be modified to generate different types of outputs (in this case, non-player-character related game assets).

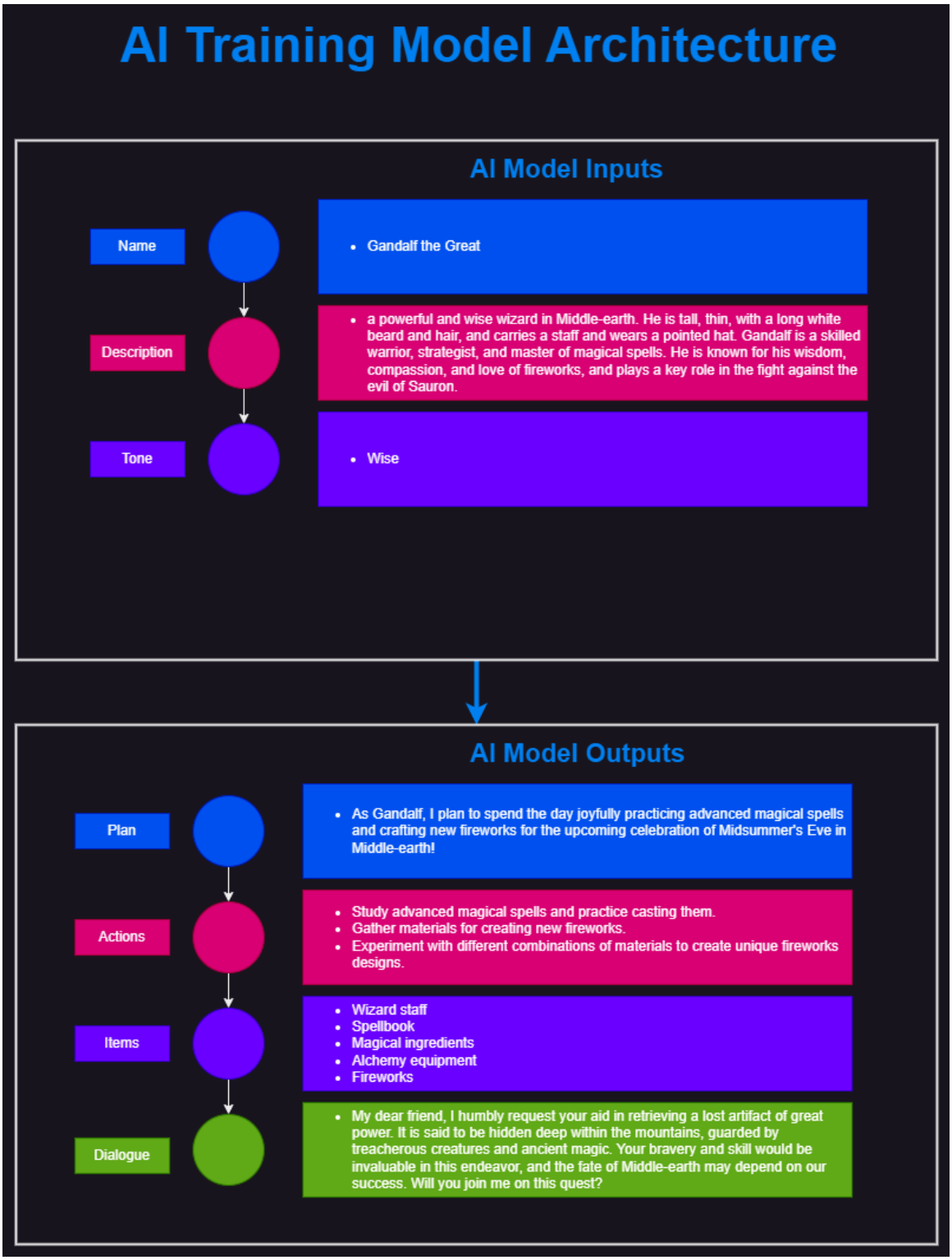


Figure 14: Example Pipeline Architecture Diagram

6 Training the Text-To-Image Generation Model

Our architecture enables the ability to generate game art alongside text-based game assets. This could range from: 2D textures, isometric game worlds, game items or pickups, and more. This section of the report discusses how we can implement this functionality into our pipeline utilising deep learning text-to-image generation models and custom training data sets.

6.1 Experimenting with Stable Diffusion's Base Model

Deep-learning text-to-image generation models have advanced exponentially over the past few years. However, the models very rarely provide quality output using their standard model training.

Figure 15 aims to showcase a brief overview of our experiments utilising base text-to-image models with no fine-tuning. Models such as DALL.E-2 and Stable Diffusion outputs are shown as examples for a simple blue potion text prompt. It highlights the lack of overall depth, and quality in the rendered outputs. Random artefacts were also prominent and indeterminate.

DALLE-2		Not as high quality as Midjourney, but definitely seems to do better with smaller prompts than Stable Diffusion. However, it is much more expensive than both. It does have an API however. (115 API calls per month ($15 USD)
Stable Diffusion		Stable diffusion often provided weird artefacts such as computer generated text in images, and often wouldn't give the desired output of small prompts. However, if you add much more detail it could suffice. It has an API and is the cheapest to run out of the 3. (999 API calls per month $9 USD)

Figure 15: Stable Diffusion and DALL.E-2 Base Model Output Tests

6.2 Fine Tuned Model Design Criteria

To solve this problem, we needed to find a way to generate higher quality outputs from simple text prompts. While we found we could generate better image generations with more detailed and crafted prompts, the added time and manual labour complexity involved on the user side was detrimental to our overall objective. The purpose of our architecture was to reduce this labour time input, so alternatives were needed and thus, researched.

Our criteria for our text-to-image model required that game assets were able to be generated from simple prompts and descriptors, matching the overall design and stylistic preferences we desired when generating image based asset renders.

Referencing research from Google and Boston University Ruiz et al. (2022), we discovered that we could utilise DreamBooth to fine tune our text-to-image diffusion models. This allowed us to customise how we generate image related game assets, using the stylistic preferences for renders we prefer with smaller text based prompts as input.

6.3 Fine Tuning the Stable Diffusion Model

To train our own model with our custom data set, we leveraged Leonardo AI. Leonardo AI provided us access to custom fine-tuned models that were trained on more specific data building on top of Stable Diffusion's open source model. However, they also provide access to a web interface and API that allows us to train our own model via DreamBooth. By leveraging our own training data, we were able to train custom fine-tuned models to generate higher quality image renders. It's important to note we can extend this concept and train multiple models for different asset types we want to create, which is useful for rendering more detailed or specific game asset types.

Figure 16 provides the image training data we utilised to create a fine-tuned model via Stable Diffusion and Leonardo AI's web interface. Training data was created using MidJourney's text-to-image model, as well as previously generated AI game art, to ensure copyright laws were not violated.

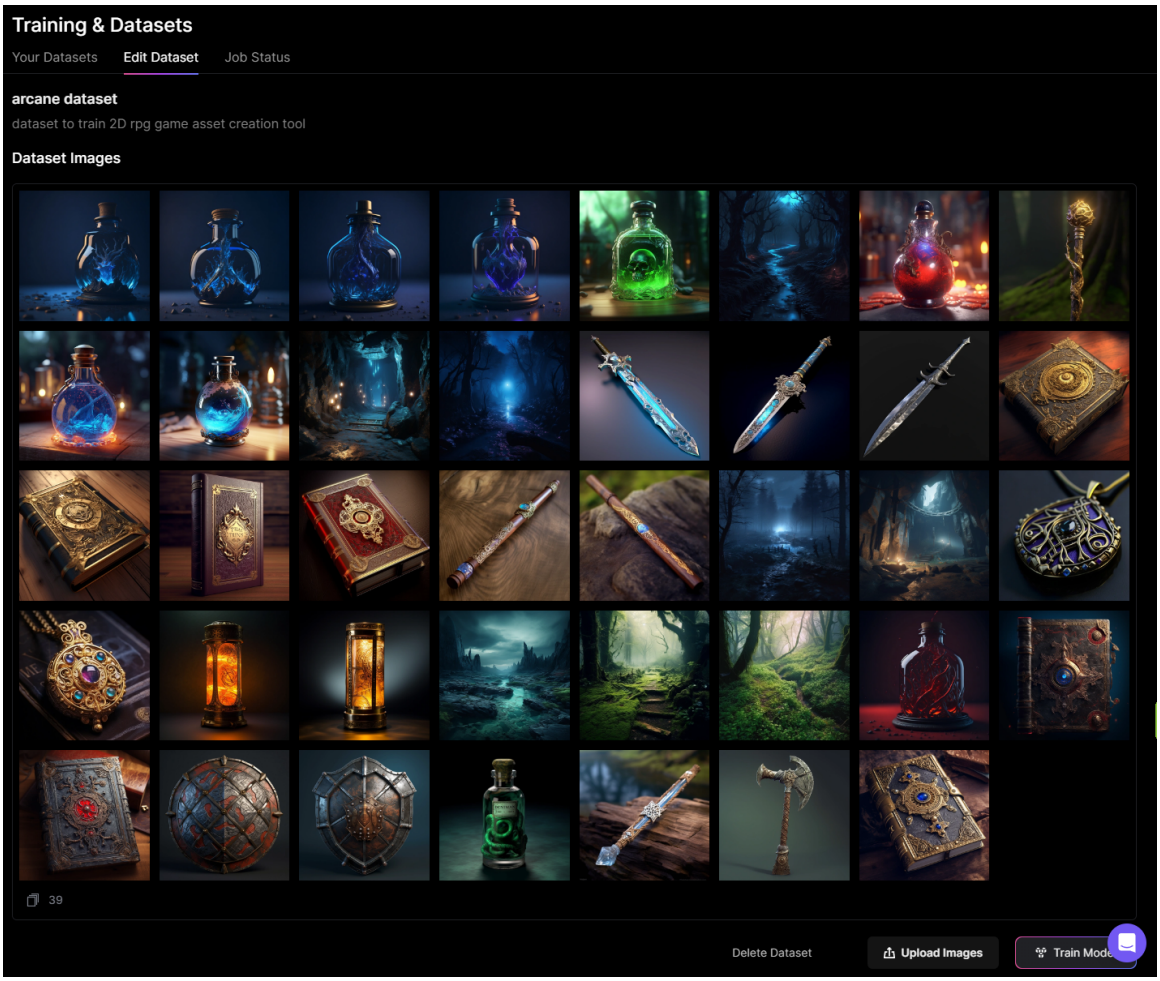


Figure 16: Fine-tuned Model Training Data

6.4 Leonardo AI API Endpoints and Model Parameters

Our model was trained on images with dimensions 512 x 512 pixels in size. The height and width of our image generation parameters were set to match, in order to ensure the best possible image generation results with our model.

Secondly, we set the guidance scale parameter to 7, as this determines how strongly our prompt is weighted. We don't want it to be too rigid, so that the model can't deviate or create interesting concepts from the prompt, but we also don't want it to lean too far from the mark of what we asked it for. We found 7-8 to be the sweet spot, but can be adjusted accordingly.

Lastly, the modelId parameter that you pass to the model determines the output of your images. We can utilise different fine-tuned models that we trained (as discussed in the Fine Tuning the Stable Diffusion Model section of this report). The modelId parameter is passed as an argument from our image generation prompts, where different prompts are created depending on the type of game asset

we wish to create. Prompts are explained in further detail within the Building the Image Generation Prompts section of this report.

Figure 17 provides details on the API endpoint parameters and headers required to generate high quality text-to-image generations using our fine-tuned Stable Diffusion models.

```python
# Get Leonardio.ai API key, User ID and Model ID from .env file
load_dotenv()
LEONARDO_AI_API_KEY = os.environ.get('LEONARDO_AI_API_KEY')
LEONARDO_AI_USER_ID = os.environ.get('LEONARDO_AI_USER_ID')


# NOTE: Create a Generation of Images

def generate_image(prompt: str, model_id: str):
    url = "https://cloud.leonardo.ai/api/rest/v1/generations"

    payload = {
    "prompt": prompt,
    "modelId": model_id,
    "presetStyle": "LEONARDO",
    "width": 512,
    "height": 512,
    "num_images": 1,
    "guidance_scale": 7,
    "public": False,
    "sd_version": "v2",
    "scheduler": "LEONARDO"
    }

    headers = {
    "Authorization": f"Bearer {LEONARDO_AI_API_KEY}",
    "content-type": "application/json",
    "accept": "application/json"
    }

    response = requests.post(url, json=payload, headers=headers)
    response_json = response.json()
    generation_id = response_json['sdGenerationJob']['generationId']

    # Give the URL enough time to populate
    time.sleep(10)
    image_url = get_image_by_id(generation_id)
    return image_url
```

Figure 17: Leonardo AI Generate Image Endpoint

6.5 Building the Image Generation Prompts

In order to for us to be able to customise how we want each item to be stylised or created we need to provide our deep-learning text to image models with prompts to curate game assets to our stylistic preferences. To do this, we created a Prompts class, where essentially different types of prompts are utilised, based on the type of asset we would like to create.

For example: for game items we might want to design a prompt to specify whether we want the item to render as a highly realistic octane render, or a 2D sprite sheet. Alternatively, we may also want to render completely different types of assets such as textures, isometrics, maps, etc. Our prompt class allows us to create and tweak our prompts, based on the specific style requirements for the game asset types developers would like to create. This was designed in a modular and low-coupled manner.

Figure 18 provides a UML diagram, illustrating the class structure and design of our prompt classes.

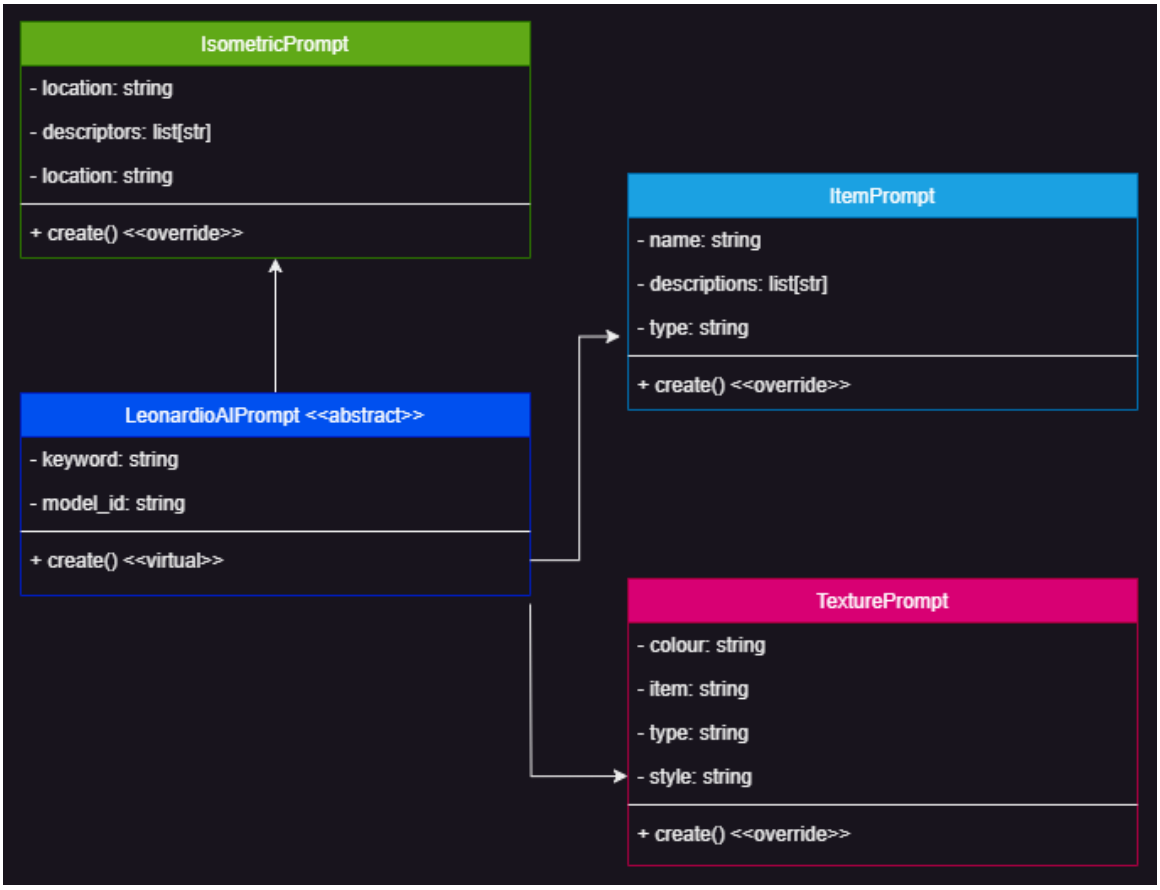


Figure 18: Prompts Class UML Diagram

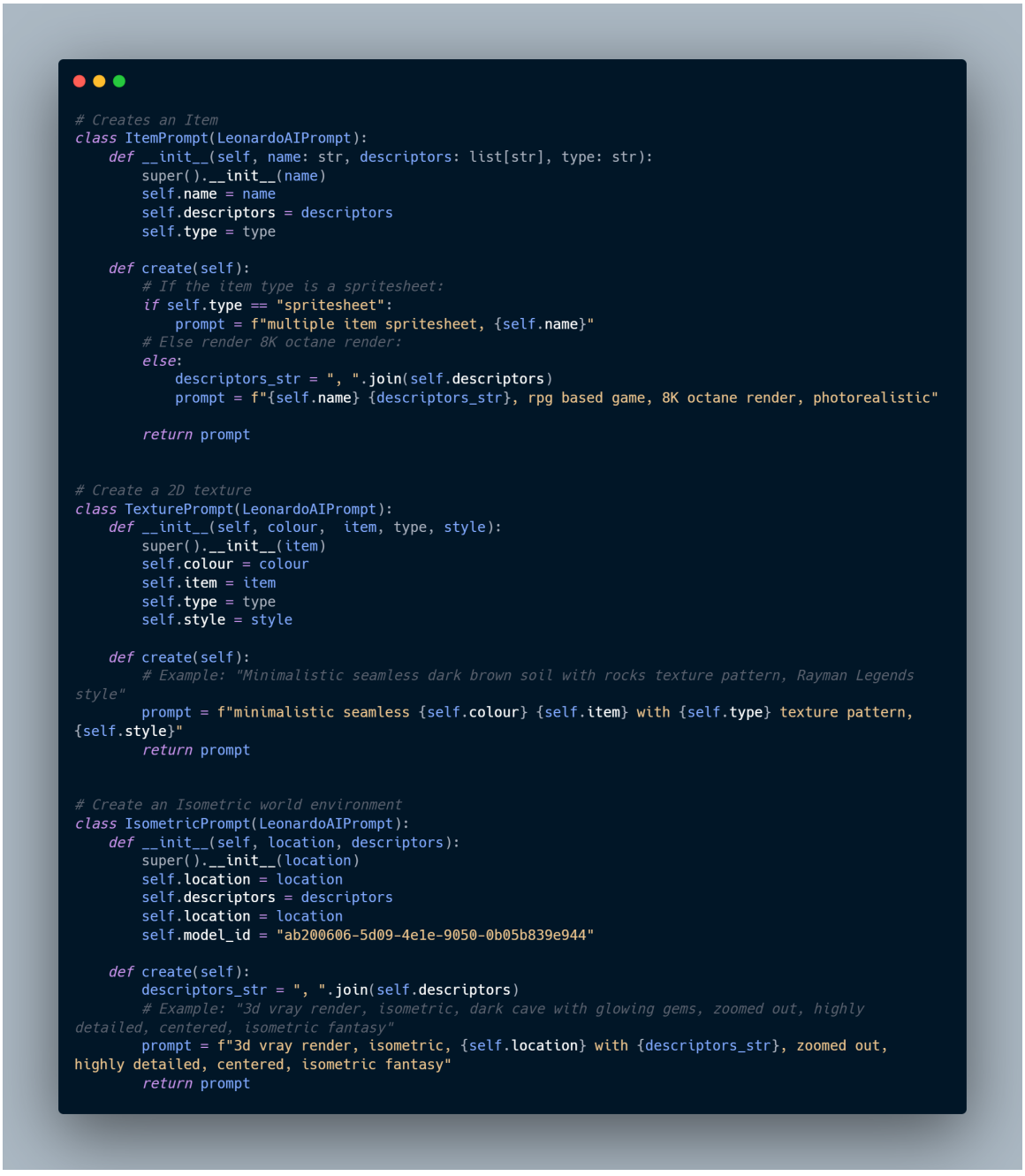
Figure 19 highlights the base class that prompts inherit from. Prompts require a model ID be passed to it, so that we can change or modify the type of text to image generation model used, to render better stylistic results for different types of items (more hyper-tuned).

```python
# Abstract base class
class LeonardoAIPrompt(ABC):
    def __init__(self, keyword: str):
        self.keyword = keyword
        self.model_id = "b820ea11-02bf-4652-97ae-9ac0cc00593d"

    def create(self):
        pass
```

Figure 19: Prompt Abstract Base Class

Figure 20 demonstrates the use of inheritance in generating or customising different prompts for different types of game assets.



```python
# Creates an Item
class ItemPrompt(LeonardoAIPrompt):
    def __init__(self, name: str, descriptors: list[str], type: str):
        super().__init__(name)
        self.name = name
        self.descriptors = descriptors
        self.type = type

    def create(self):
        # If the item type is a spritesheet:
        if self.type == "spritesheet":
            prompt = f"multiple item spritesheet, {self.name}"
        # Else render 8K octane render:
        else:
            descriptors_str = ", ".join(self.descriptors)
            prompt = f"{self.name} {descriptors_str}, rpg based game, 8K octane render, photorealistic"

        return prompt


# Create a 2D texture
class TexturePrompt(LeonardoAIPrompt):
    def __init__(self, colour,  item, type, style):
        super().__init__(item)
        self.colour = colour
        self.item = item
        self.type = type
        self.style = style

    def create(self):
        # Example: "Minimalistic seamless dark brown soil with rocks texture pattern, Rayman Legends
style"
        prompt = f"minimalistic seamless {self.colour} {self.item} with {self.type} texture pattern,
{self.style}"
        return prompt


# Create an Isometric world environment
class IsometricPrompt(LeonardoAIPrompt):
    def __init__(self, location, descriptors):
        super().__init__(location)
        self.location = location
        self.descriptors = descriptors
        self.location = location
        self.model_id = "ab200606-5d09-4e1e-9050-0b05b839e944"

    def create(self):
        descriptors_str = ", ".join(self.descriptors)
        # Example: "3d vray render, isometric, dark cave with glowing gems, zoomed out, highly
detailed, centered, isometric fantasy"
        prompt = f"3d vray render, isometric, {self.location} with {descriptors_str}, zoomed out,
highly detailed, centered, isometric fantasy"
        return prompt
```

Figure 20: Different Game Asset Creation Prompts such as Items, Textures and Isometrics

6.6 Generating Assets from the Text-To-Image Prompts

Within the "Building the Game Asset Creation Pipeline" section of this report, we demonstrated how to chain together multiple game assets, primarily for text based asset generation. In order to showcase how text-to-image assets can be rendered via the pipeline architecture, we created a secondary Quest Creation Pipeline. This allows us to showcase how to generate quest items in the form of image renders via our fine-tuned models.

Figure 21 demonstrates how to invoke the Leonardo API endpoint within the staging process of the pipeline, to generate images for different asset types. In this scenario, we're creating game quest related items utilising our ItemPrompt class. The GPT-3.5 model is leveraged to describe or think of the item descriptors based on the quest objectives or storyline. This data is then passed to our fine-tuned text-to-image model as a prompt, before generating an image output.

```python
# Chaining pipeline stage 2
def stage_2(self):
    # Generate Asset Images

    # Step 1: Create Item Assets
    for item in self.assets['items_details']:
            asset = ItemPrompt(item['description'], descriptors="", type="quest")
            prompt = asset.create()

            # Generate the asset's Image using Leonardo.ai API endpoint
            image = generate_image(prompt, asset.model_id)

            # Grab the Image URL from the response
            image_url = image['generations_by_pk']['generated_images'][0]['url']

            # Add the Item Image URL to the list of items_details in the JSON file
            item['url'] = image_url

            # Write the updated data back to the Items data file
            self.write_data_to_file(self.assets)


# Execute the pipeline stages
def run(self):
    # Run Chain 1
    self.stage_1()
    # Run Chain 2
    self.stage_2()
    # Display message once complete
    self.display_message()
```

Figure 21: Example of Invoking Image Generation Endpoint within the Pipeline

7 Conclusion

In conclusion, our constraint bootstrapping architecture provides a framework for automatically generating 2D game assets utilising artificial intelligence and deep learning models. Leading industry research and recent innovations within the artificial intelligence space has provided a new avenue for domain specific use cases to speed up and reduce costs for game related assets. Our research aims to illustrate how chaining artificially generated output can be utilised to automate the generation of multiple game assets, albeit the need for some initial upfront development time due to modifying parameters, creating domain specific pipelines and prompts. This is to ensure that output is of high quality, and better matches the stylistic preferences of game developers or game related projects. Our architecture serves as a base concept which is modular in nature, and can be extended upon.

Appendices

Templates

```python
from langchain.prompts import PromptTemplate

"""Prompt Templates"""

# NOTE: Chain 1: Sample inputs: {theme: "skyrim"}, {details: "set in the dark forests of Gloomhaven.
The forest is looming with darkness and mystery with a river that glows blue at night for some reason.
It evokes a sense of danger and foreboding, complimented by a misty atmosphere."}

# Create game storyline based on the initial concept theme and game scene details.
# Inputs: {"theme", "details"}
# Outputs: {"storyline"}
storyline_template = PromptTemplate(
    input_variables=["theme", "details"],
    template="create a short storyline for an rpg based game based around this concept theme: {theme}.
Then modify the storyline based on these provided additional details regarding the game scene:
{details} IMPORTANT: DO NOT include the Original storyline in your response, only provide the modified
storyline.",
)

# Create 2 NPC Characters based on the game storyline.
# Inputs: {"storyline"}
# Outputs: {"characters"}
characters_template = PromptTemplate(
    input_variables=["storyline"],
    template="from the game storyline: {storyline}, create 2 fictional game characters (NPCs). List
them and a brief character description (1-2 sentences each).",
)



# NOTE: Chain 2: Grabs inputs {"storyline", "characters"} from Chain 1 output.

# Create character backstories.
# Inputs: {"storyline", "characters"}
# Outputs: {"backstories"}
backstories_template = PromptTemplate(
    input_variables=["storyline", "characters"],
    template="based on these character descriptions: {characters}, describe their character backstory,
including their past, what's happened to them, what they are trying to do, etc. Important: Take into
consideration the game storyline {storyline} and the other NPC characters in the game to form your
answer."
)



# NOTE: Chain 3: Grabs inputs {"backstories"} from Chain 2 ouput, and {"storyline"} from Chain 1
output.
# Create character specific quests.
# Inputs: {"backstories", "storyline"}
# Outputs: {"quests"}
quests_template = PromptTemplate(
    input_variables=["backstories", "storyline"],
    template="based on these character backstories: {backstories}, as well as the game world storyline:
{storyline}, provide a quest that they may ask a player to embark on in the game world. Important: make
sure it utilises the context of the game world and their backstory/descriptions."
)
```

Figure 22: AutoPilot Templates 1-3

```
# NOTE: Chain 4: Grabs the inputs {"quests"} from Chain 3 Output, and {"backstories"} from Chain 2
output.
# Create branching-dialogue trees for each character to help players embark on the quest provided.
# Inputs: {"quests", "backstories"}
# Outputs: {"dialogue"}
dialogue_template = PromptTemplate(
    input_variables=["quests", "backstories"],
    template="for each of these quests: {quests}, provide some branching-dialogue trees to help a
player interact with the character to complete the quest. Make sure the dialogue of the character uses
their personality tone which can be drawn from their character backstory: {backstories}"
)



# NOTE: Chain 5: Grabs the inputs {"dialogue"} from Chain 4 output, and {"quests"} from Chain 3 output.

# Create branching-dialogue trees for each character to help players embark on the quest provided.
# Inputs: {"dialogue", "quests"}
# Outputs: {"items"}
items_template = PromptTemplate(
    input_variables=["dialogue", "quests"],
    template="based on the actions the player may need to take to complete the given quests: {quests}
successfully (guided by the dialogue: {dialogue}), what items would you need to complete these tasks?
For example: if you're a wizard, you might require a staff item to pass through Middle earth, etc. Just
state the items in dot-point form. Don't go into details. Split all items separately without mention of
the tasks. For each of these items, make sure only one item is outputted per line. Secondly make it a
specific item. For example: mana potion, redstone dust, oak staff, etc. Don't tell me why the item is
there and make sure only one item per line (Very important). List 2 items."
)

# Inputs: {"items"}
# Outputs: {"item_descriptions"}
item_descriptions_template = PromptTemplate(
    input_variables=["items"],
    template="describe the items: {items} with intricate details, that provides a vivid image of how
the item will look based on descriptors. For example: A magical potion with blue liquid in it might be:
bottles with glow spells on top, in the style of realistic and hyper-detailed renderings, enchanting
realms, uhd image, detailed character illustrations, eerily realistic, hyper-realistic water, gold and
blue."
)



# NOTE: Chain 6: Converts dialogue text into JSON format for branching-dialogue tree exporting.
# Convert the dialogue into a JSON branching-dialogue tree structure.
# Inputs: {"dialogue"}
# Outputs: {"dialogue_tree"}
dialogue_tree_template = PromptTemplate(
    input_variables=["dialogue"],
    template="convert these branching-dialogue trees into a JSON file: {dialogue} Important: Don't
include the initial sentence like: Here is the JSON file for the branching dialogue trees:"
)
```

Figure 23: AutoPilot Templates 4-6

Chains

```python
import os
from dotenv import load_dotenv
# from langchain.llms import OpenAI
from langchain.chat_models import ChatOpenAI
from langchain.chains import LLMChain
from .templates import storyline_template, characters_template, backstories_template, quests_template,
dialogue_template, items_template, item_descriptions_template, dialogue_tree_template

# get ChatGPT API key from .env file
load_dotenv()
OPENAI_API_KEY = os.getenv("OPENAI_API_KEY")

# LLMS

#OpenAI
llm = ChatOpenAI(temperature=0.7, model_name="gpt-3.5-turbo")
# API model costs: https://openai.com/pricing

# NOTE: Chain 1

storyline_chain = LLMChain(
    llm=llm,
    prompt=storyline_template,
    verbose=True,
    output_key="storyline"
)

characters_chain = LLMChain(
    llm=llm,
    prompt=characters_template,
    verbose=True,
    output_key="characters"
)

# NOTE: Chain 2

backstories_chain = LLMChain(
    llm=llm,
    prompt=backstories_template,
    verbose=True,
    output_key="backstories"
)

# NOTE: Chain 3

quests_chain = LLMChain(
    llm=llm,
    prompt=quests_template,
    verbose=True,
    output_key="quests"
)

# NOTE: Chain 4

dialogue_chain = LLMChain(
    llm=llm,
    prompt=dialogue_template,
    verbose=True,
    output_key="dialogue"
)

# NOTE: Chain 5

items_chain = LLMChain(
    llm=llm,
    prompt=items_template,
    verbose=True,
    output_key="items"
)

item_descriptions_chain = LLMChain(
    llm=llm,
    prompt=item_descriptions_template,
    verbose=True,
    output_key="item_descriptions"
)

# NOTE: Chain 6

dialogue_tree_chain = LLMChain(
    llm=llm,
    prompt=dialogue_tree_template,
    verbose=True,
    output_key="dialogue_tree"
)
```


Figure 24: AutoPilot Chains

References

Park, J. S., O'Brien, J. C., Cai, C. J., Morris, M. R., Liang, P. & Bernstein, M. S. (2023), 'Generative agents: Interactive simulacra of human behavior'.

Ruiz, N., Li, Y., Jampani, V., Pritch, Y., Rubinstein, M. & Aberman, K. (2022), 'Dreambooth: Fine tuning text-to-image diffusion models for subject-driven generation'.

ShivamShrirao (2023), 'Dreambooth stable diffusion'.
URL: *https://github.com/ShivamShrirao/diffusers*