# Architecting an AI Game Master: A Technical Guide to Customizing a Local LLM for the Mothership RPG

## Executive Summary: Architecting Your AI Game Master

This document presents a comprehensive technical framework for transforming a locally-hosted Large Language Model (LLM) into a specialized, knowledgeable AI Game Master—or "Warden," in the game's parlance—for the *Mothership* Sci-Fi Horror Roleplaying Game. The project commences from a state where the user has successfully deployed the Tohur/Natsumura Storytelling RPG Llama 3.1 model via Ollama, establishing a foundation for local, offline generation. The objective is to evolve this generalist RPG model into a domain-specific expert capable of managing game rules, narrating scenarios, and embodying the unique tone of the *Mothership* universe.

The core technical challenge lies in the effective injection and integration of domain-specific knowledge. This knowledge is not monolithic; it encompasses both explicit, factual data—such as game mechanics, equipment statistics, and module-specific lore—and implicit, behavioral patterns, which include the narrative style, pacing, and decision-making faculties of an effective *Mothership* Warden. To address this multifaceted challenge, this report details two primary technical methodologies: Retrieval-Augmented Generation (RAG) and Supervised Fine-Tuning (SFT).

While each approach offers distinct advantages, a singular choice presents significant trade-offs. RAG excels at providing factual accuracy from a dynamic knowledge base but fails to teach the model a specific persona. Conversely, SFT is unparalleled for shaping a model's behavior and style but is computationally expensive and creates a static knowledge state. Therefore, this report advocates for a sophisticated hybrid architecture as the optimal solution. In this model, RAG serves as the primary mechanism for real-time access to the extensive corpus of *Mothership* rules, modules, and tables—defining *what the AI knows*. Concurrently, a targeted SFT process is employed to shape the AI's core persona, its narrative

voice, and its intrinsic understanding of the game's high-stakes, horror-centric mechanics—defining *how the AI acts*. This hybrid strategy, which aligns with advanced industry practices, leverages the strengths of each technique to create a robust, adaptable, and highly capable AI Warden.[1]

The project is structured as a four-phase roadmap, guiding the developer from strategic planning through to iterative refinement:

1. **Strategic Decision & Corpus Assembly:** This initial phase focuses on a rigorous analysis of the available architectural pathways and addresses the critical, labor-intensive task of collecting, cleaning, and structuring the proprietary *Mothership* game data into a machine-readable format.
2. **Technical Implementation:** This phase provides detailed, step-by-step instructions for building the necessary technical infrastructure, with complete implementation guides for both the RAG pipeline and the SFT process.
3. **Persona & Behavior Definition:** This phase details the final layer of customization, utilizing Ollama's native Modelfile capabilities to bake in the AI Warden's core personality, directives, and operational parameters.
4. **Evaluation & Iteration:** The final phase establishes a qualitative framework for testing the AI GM's performance, moving beyond simple functionality checks to assess its quality as a game facilitator and outlines a continuous loop for refinement.

By following this structured approach, a developer can systematically engineer an AI companion tool that not only generates text but also functions as a coherent, knowledgeable, and thematically appropriate Game Master for the *Mothership* RPG.

## The Core Decision: Choosing Your Knowledge Integration Strategy

The foundational architectural decision for this project is determining the method by which the LLM will acquire and utilize knowledge specific to the *Mothership* RPG. This choice is not merely a technical implementation detail; it dictates the entire project's workflow, resource requirements, and long-term maintenance strategy.[3] The selection of an integration strategy impacts everything from the nature of the data preparation effort to the computational budget required for both development and deployment. Two primary methodologies, Retrieval-Augmented Generation (RAG) and Supervised Fine-Tuning (SFT), present distinct paradigms for this task. A third option, a hybrid of the two, offers a sophisticated path that mitigates the weaknesses of each individual approach.

## Retrieval-Augmented Generation (RAG): The "Open Book" Warden

**Concept:** RAG is a technique that dynamically augments an LLM's knowledge at the time of inference. Instead of relying solely on the information encoded in its parameters during training, the model is given access to an external, up-to-date knowledge base.[1] When a query is received, a retrieval system first searches this knowledge base for relevant information. This retrieved data is then packaged with the original query and passed to the LLM as part of an expanded prompt. The LLM uses this just-in-time information to formulate its response.[4] For the AI GM, this is analogous to a human Warden pausing the game to consult the

*Player's Survival Guide* or a specific adventure module to confirm a rule or detail. The model does not "learn" the information in a permanent sense; it references it as needed.

**Strengths:**

- **Reduces Hallucinations:** By grounding the model's responses in factual data retrieved from a trusted source (the game's rulebooks), RAG dramatically mitigates the risk of the LLM "hallucinating"—inventing plausible but incorrect rules, monster statistics, or lore details.[1] This is paramount for an application that must function as a rules arbiter.
- **Knowledge Agility:** The external knowledge base is decoupled from the LLM itself. This allows for effortless updates. When a new third-party *Mothership* module is released, it can be added to the AI's knowledge simply by processing its PDF and adding it to the vector store. No model retraining is necessary, making the system highly adaptable and scalable.[2]
- **Traceability and Verifiability:** Because the retrieved context is an explicit part of the generation process, responses can be traced back to their source documents. This provides a clear audit trail, which is invaluable for debugging the system and allowing users to verify the source of a given rule interpretation.[4]
- **Lower Computational Barrier to Entry:** RAG avoids the computationally intensive and often costly process of model training. The primary workload is in the initial setup of the data pipeline and vector store, which is generally more accessible than fine-tuning.[1]

**Weaknesses:**

- **Architectural Complexity:** A functional RAG system is not a single component but a multi-stage pipeline. It requires the orchestration of a document loader, a text chunker, an embedding model, a vector database for storage and retrieval, and the LLM itself. Integrating and optimizing these components requires significant engineering effort.[5]
- **Inference Latency:** The retrieval step introduces additional processing time for every query. The system must embed the query, search the vector store, and process the retrieved documents before the LLM can even begin generating a response. This can

lead to higher latency compared to a self-contained model, which may impact the flow of a real-time game session.[8]

- **Provides Knowledge, Not Behavior:** RAG is exceptionally effective at providing the LLM with facts. However, it is fundamentally incapable of teaching the model a specific *style*, *personality*, or a nuanced way of interpreting information. It can tell the model the rule for a Panic Check, but it cannot teach it how to describe the outcome with the appropriate sense of sci-fi horror dread. It provides an "open book" but doesn't teach the art of storytelling.[2]

## Supervised Fine-Tuning (SFT): The "Studied" Warden

**Concept:** SFT is a process that adapts a pre-trained LLM by continuing its training on a smaller, curated dataset of examples. For a conversational AI, this dataset typically consists of high-quality instruction-response pairs.[10] During this process, the internal weights and parameters of the model are adjusted to minimize the difference between its generated responses and the "correct" responses in the dataset. The model effectively

*learns* the new information and behaviors, embedding them directly into its neural architecture.[4] This is analogous to a human Warden who has spent weeks studying the rulebooks, internalizing the mechanics, and developing their own unique GMing style through practice.

**Strengths:**

- **Teaches Behavior and Style:** SFT is the premier method for shaping an LLM's persona. By training the model on examples of high-quality *Mothership* narration, rules adjudication, and NPC dialogue, it can learn to replicate that specific tone and style. This is how the AI transitions from a generic storyteller to a gritty, horror-focused Warden.[3]
- **Embeds Foundational Knowledge:** For core, static, and frequently accessed information—such as the basic action resolution mechanic, the definitions of the four character classes, or the Stress and Panic rules—embedding this knowledge directly into the model via fine-tuning can be more efficient than retrieving it from a database for every single query.[2]
- **Lower Inference Latency:** Once the fine-tuning process is complete, the result is a single, self-contained model. During inference, there is no external retrieval step, which can lead to faster response generation compared to a complex RAG pipeline.[8]

**Weaknesses:**

- **High Computational Cost and Resource Requirements:** Full-parameter fine-tuning is extremely resource-intensive, requiring multiple high-end GPUs and vast amounts of

memory, placing it outside the reach of most individuals. While more accessible Parameter-Efficient Fine-Tuning (PEFT) methods like LoRA and QLoRA exist, they still demand significant GPU resources (often a dedicated cloud instance or a high-end local GPU) and considerable training time.[12]

- **Risk of Catastrophic Forgetting:** The process of adjusting the model's weights to learn new information can sometimes interfere with or erase knowledge it previously possessed. If not managed carefully, fine-tuning for *Mothership* could degrade the model's general language capabilities or the storytelling skills it already has.[12]
- **Knowledge Brittleness and Static Nature:** The model's knowledge is frozen at the moment its training concludes. To incorporate a new rulebook, errata, or adventure module, the entire, resource-intensive fine-tuning process must be repeated. This makes the system rigid and difficult to maintain.[4]
- **Dependence on High-Quality Curated Datasets:** The performance of a fine-tuned model is entirely dependent on the quality of its training data. Creating a comprehensive, accurate, and stylistically consistent dataset of instruction-response pairs is a significant, labor-intensive, and expert-driven task. A poorly constructed dataset will result in a poorly performing model.[1]

## The Hybrid Vanguard: The Professional's Choice

**Concept:** A hybrid architecture seeks to combine the strengths of both RAG and SFT while mitigating their respective weaknesses.[1] The process involves two distinct stages:

1. **Style-Tuning:** The base LLM is first fine-tuned using a relatively small, high-quality dataset. The goal of this stage is not to teach the model the entire corpus of *Mothership* knowledge, but rather to teach it the *style*, *persona*, and *structure* of the game. It learns how to interpret player requests in the context of *Mothership*, how to structure a ruling, and how to generate narrative descriptions in the correct sci-fi horror tone.
2. **RAG Integration:** This style-tuned model then becomes the generative component within a standard RAG pipeline. The RAG's knowledge base contains the full, detailed corpus of rulebooks, modules, and tables.

Synergistic Benefits:
This architecture creates a powerful synergy. The fine-tuned model is inherently better at understanding the context of a Mothership game. When a player asks, "The alien lunges at me, what do I do?", the tuned model understands this is a moment of physical threat and is better equipped to formulate a relevant, targeted query to the RAG system, perhaps searching for "creature attacks" or "evasion rules." The RAG system, in turn, provides the specific, factual context (e.g., the creature's stat block and the rules for Body saves), which prevents the specialized model from hallucinating and ensures its stylistically appropriate response is

also mechanically accurate. This architecture effectively separates the concerns of "behavioral competence" (handled by fine-tuning) from "factual knowledge" (handled by RAG), resulting in a system that is more robust, maintainable, and ultimately more capable than either approach in isolation.

## Strategic Recommendation for the Mothership Companion

For this project, a phased approach is recommended, prioritizing immediate functionality while building towards the more sophisticated hybrid architecture.

1. **Phase I: RAG-First Implementation.** The initial development effort should focus on building a complete RAG pipeline using the base Tohur/Natsumura model. This provides the most immediate value by granting the LLM access to the necessary rulebooks and lore. It is the fastest path to a functional prototype that can answer specific, fact-based questions about the game.
2. **Phase II: Style-Focused Fine-Tuning.** Once the RAG system is operational, the second phase should involve creating a small, high-quality dataset to perform a light, style-focused fine-tuning of the base model. The resulting specialized model would then replace the base model in the RAG pipeline. This enhancement will refine the AI's personality to better match the gritty tone of *Mothership* and improve its understanding of core game loops, making its interactions with both the user and the RAG system more intelligent and thematically appropriate.

This phased strategy allows for incremental development and testing, delivering a useful tool early while providing a clear path for significant enhancement over time.

| Feature | Retrieval-Augmented Generation (RAG) | Supervised Fine-Tuning (SFT) | Hybrid Approach (SFT + RAG) |
|---|---|---|---|
| **Core Concept** | Augments prompts with externally retrieved, real-time data. | Adjusts internal model weights by training on a curated dataset. | A style-tuned model uses a RAG system for factual data. |
| **Computational Cost (Training)** | Low (Primarily data indexing). | High (Requires significant GPU resources, even with PEFT). | High (Requires SFT process first). |
| **Computational** | Medium (Retrieval | Low | Medium (Retrieval |

| | | | |
|---|---|---|---|
| **Cost (Inference)** | step adds latency). | (Self-contained model). | step adds latency). |
| **Implementation Complexity** | Medium (Requires building a multi-stage data pipeline). | High (Requires expertise in training loops, PEFT, and model conversion). | Very High (Combines the complexity of both approaches). |
| **Data Prep Effort** | Medium (Requires cleaning and chunking of source documents). | Very High (Requires manual creation of high-quality instruction-respons e pairs). | Very High (Requires both structured documents for RAG and a curated dataset for SFT). |
| **Knowledge Agility** | High (Knowledge base can be updated without retraining). | Low (Requires complete retraining to add new knowledge). | High (RAG component allows for dynamic knowledge updates). |
| **Hallucination Risk** | Low (Responses are grounded in retrieved facts). | Medium-High (Model can still invent details if not explicitly trained). | Low (Grounded by RAG, with SFT potentially reducing nonsensical queries). |
| **Style/Behavior Learning** | Very Low (Model uses its base personality). | High (Primary method for teaching persona, tone, and style). | High (SFT component is dedicated to teaching behavior). |
| **Traceability** | High (Responses can be traced to source documents). | Low (Internal decision-making is opaque). | High (Responses can be traced to RAG-retrieved sources). |
| **Ideal Use Case for AI GM** | Answering specific rules questions; looking up module details. | Embodying the Warden persona; generating atmospheric | A comprehensive AI Warden that is both knowledgeable and stylistically |

| | | | |
|---|---|---|---|
| | | narration. | appropriate. |

# Phase 1: Assembling the Knowledge Corpus

The success of any knowledge-intensive AI project is determined not by the sophistication of the model alone, but by the quality and structure of the data it is given. For the AI Warden project, this phase of assembling the knowledge corpus is the most critical and labor-intensive stage. The outcome of this phase will directly constrain and define the ultimate capabilities of the final system.

## The Mothership Data Challenge: The Absence of an SRD

A significant challenge specific to the *Mothership* RPG is the absence of a System Reference Document (SRD). In the tabletop RPG industry, an SRD is a document provided by the publisher that contains the core, open-licensed mechanics of the game. Games like Dungeons & Dragons provide an extensive SRD, which allows third-party creators to legally use and reproduce the base game rules in their own products.[17] This creates a machine-readable, legally accessible foundation of game mechanics.

*Mothership*, however, does not currently offer such a document. The core rules, lore, and content are contained within proprietary, commercially available books, often presented in a highly stylized and graphical format designed for human readers, not machine parsing.[18] This fundamental difference means that all necessary data must be sourced from these proprietary materials and manually processed into a structured format.

*A Note on Legal and Ethical Considerations:* This project, by necessity, involves the extraction of copyrighted material. It is imperative that this work is undertaken for personal, non-commercial use only. The user must legally purchase all official rulebooks and modules. The distribution of any resulting fine-tuned model or the curated dataset itself would likely constitute a copyright infringement against Tuesday Knight Games and the various third-party publishers. This guide is provided for the purpose of personal project development and research.

## Step 1: Sourcing the Raw Data

The first step is to acquire a comprehensive library of *Mothership* materials. The goal is to create a knowledge base that is as complete as possible, mirroring the resources a dedicated human Warden would have at their table.

**Official Core Rulebooks (1st Edition):** These are the highest-priority data sources, forming the bedrock of the AI's understanding of the game.

- ***Player's Survival Guide:*** Contains all fundamental player-facing rules, including character creation, skill checks, combat, and the crucial Stress and Panic systems.[19]
- ***Warden's Operations Manual:*** Provides the Game Master's rules, procedures for running sessions, creating scenarios, and tools for campaign preparation.[19]
- ***Unconfirmed Contact Reports:*** The game's bestiary, containing descriptions, statistics, and behaviors for various alien creatures and horrors.[19]
- ***Shipbreaker's Toolkit:*** Details the rules for spacecraft, including operation, maintenance, and ship-to-ship combat.[19]

**Official Modules and Campaigns:** These provide the AI with concrete adventure content, locations, NPCs, and narrative structures.

- **Essential Starter Modules:** *Another Bug Hunt* (the official introductory adventure) and *The Haunting of Ypsilon 14* (a classic starting point) are critical for teaching the AI how a typical *Mothership* session unfolds.[19]
- **Major Campaign Modules:** Acclaimed adventures like *Dead Planet*, *A Pound of Flesh*, and *Gradient Descent* offer rich, complex settings and long-form narrative examples that are invaluable for training.[22]

**Third-Party and Community Content:** The vibrant *Mothership* community has produced a vast amount of third-party content that significantly expands the game world. Incorporating this material will give the AI Warden a much broader range of knowledge.

- **Major Publications:** Anthologies and campaign settings like *Hull Breach Vol. 1* or *Cloud Empress* represent high-quality, professionally produced content that can be treated as semi-official.[23]
- **Community-Generated Resources:** Wikis (such as 1d6chan), forums (like the official r/MothershipRPG subreddit), and fan-made cheat sheets (e.g., The Alexandrian's System Cheat Sheet) are invaluable.[25] These sources often distill the rules into concise, practical summaries and capture the community's consensus on interpreting ambiguous mechanics, providing a different and highly useful form of data.

## Step 2: Data Extraction and Cleaning

With the source materials gathered (primarily in PDF format), the next step is to extract the raw text and clean it for machine processing. This is a non-trivial task due to the artistic layout of the sourcebooks.

Methodology:
The primary method for data extraction will be programmatic, using Python libraries such as PyPDF2 or, more effectively, pdfplumber, which is better at handling complex layouts. For web-based resources like wikis, web scraping libraries like BeautifulSoup or Scrapy will be necessary.
**Challenges and Solutions:**

- **Complex Layouts:** *Mothership* books feature multi-column text, sidebars, inset boxes, and significant graphical elements that can corrupt automated text extraction, resulting in jumbled or out-of-order text.
    - **Solution:** After an initial automated pass, a significant manual cleaning and re-ordering process will be required. This involves reading through the extracted text and correcting flow, removing page headers/footers, and fixing line breaks.
- **Tables and Charts:** Critical game data, such as weapon statistics, armor values, panic effects, and character creation flowcharts, are presented in tables that are nearly impossible to extract accurately with automated tools.[22]
    - **Solution:** This data must be manually transcribed. This is a meticulous but non-negotiable step for ensuring the mechanical accuracy of the AI. The transcribed data should be saved in a structured, machine-readable format like CSV or JSON. For example, the weapon chart from the *Player's Survival Guide* should be converted into a CSV file with columns for Weapon, Cost, Damage, Crit_Effect, Range_Short, Range_Medium, Range_Long, Shots, and Special_Properties.

## Step 3: Structuring the Corpus for RAG

For the Retrieval-Augmented Generation pipeline, the goal is to create a searchable library of discrete, semantically coherent information chunks.

Chunking Strategy:
The cleaned, full-text documents should be processed using a text splitter. The RecursiveCharacterTextSplitter from the LangChain library is highly recommended for this task.29 It attempts to split text along semantic boundaries (paragraphs, sentences) before resorting to fixed-character cuts.

- **Chunk Size:** A relatively small chunk size, such as 250 to 500 characters, is advisable. This ensures that when the system retrieves a chunk, it is highly specific to the query and contains minimal irrelevant information.
- **Chunk Overlap:** A small amount of overlap between chunks (e.g., 50 characters) helps to preserve semantic context at the boundaries of each chunk, preventing important information from being split awkwardly across two separate chunks.[30]

Metadata Enrichment:
Simply chunking the text is insufficient. Each chunk must be enriched with metadata to enable more powerful and filtered retrieval. During the chunking process, each chunk should be tagged with a dictionary of metadata, including:

- source_document: e.g., "Player's Survival Guide 1e"
- source_page: The page number from the original PDF.
- section: e.g., "Violent Encounters," "Stress & Panic," "Ship-to-Ship Combat."
- keywords: A list of relevant game terms, e.g., ["wounds", "damage", "armor points", "healing"].

This metadata allows the RAG system to perform filtered searches, such as "find information about 'wounds' only within the 'Player's Survival Guide'."

## Step 4: Structuring the Corpus for Fine-Tuning

For Supervised Fine-Tuning, the data requirements are fundamentally different. The goal is not to create a library of facts, but a curriculum of examples that teach the model *how to behave*. This dataset must be meticulously crafted, as its quality will directly determine the personality and competence of the AI Warden.[15]

Dataset Format:
The dataset should be a JSON Lines (.jsonl) file. Each line in the file is a separate JSON object representing a single training example. For conversational fine-tuning, this object typically contains a list of messages, each with a "role" (system, user, assistant) and "content".32 The prompt structure must adhere to the specific format expected by the base model, which for Llama 3.1 involves special tokens like
<|begin_of_text|>, <|start_header_id|>, and <|eot_id|>.[34]

Content Categories for the Fine-Tuning Dataset:
The dataset should be a curated collection of high-quality examples covering the primary functions of a Warden. Quality is far more important than quantity; a few hundred excellent examples are better than thousands of mediocre ones.36

1. **Rules Interpretation and Adjudication:** These examples teach the model how to

process a player's action, identify the relevant rule, and state the outcome.
    - **Instruction (user):** "My character is a Marine with a Combat stat of 55. I want to shoot the security drone with my SMG. The range is 15m. What do I do?"
    - **Response (assistant):** "Okay, that's a Combat Check. The SMG's medium range is 75m, so you have no range penalty. To hit the drone, you need to roll a d100 and get 55 or less. Let me know what you roll. If you hit, you'll deal 4d10 damage." (This response is synthesized from rules in [22]).
2. **Narrative Style and Tone:** These examples teach the model the specific literary voice of *Mothership*. The content should be grim, tense, and evocative of sci-fi horror.
    - **Instruction (user):** "We're entering the mess hall of the derelict freighter. Describe what we see."
    - **Response (assistant):** "The air is stale, thick with the smell of spoiled rations and something metallic underneath. Trays are scattered across the floor, some still holding desiccated, unrecognizable food. A single emergency light strobes erratically, casting long, dancing shadows. On one of the tables, a datapad screen is cracked, but still glowing faintly."
3. **GM Action and Procedure Generation:** These examples teach the model the procedural flow of the game, prompting players for necessary actions.
    - **Instruction (user):** "I failed my Fear Save when the creature appeared!"
    - **Response (assistant):** "You gain 1 Stress, bringing your total to [current stress + 1]. Now you must make a Panic Check. Roll the d20 Panic Die and add your current Stress total. Tell me the result, and I will consult the Panic Effect Table." (This response is based on the core game loop described in [18]).
4. **Using Game Tables:** These examples teach the model how to respond to requests to use the game's many random tables.
    - **Instruction (user):** "I'm searching the corpse of the scientist. Do I find anything interesting? Roll on the D100 Trinkets table."
    - **Response (assistant):** "Searching the pockets of the scientist's stained lab coat, you find a worn-out keycard for a room you haven't seen yet and a small, silver locket containing a faded, holographic picture of a child." (The trinket is an example from the table in [28]).

Creating this dataset is a creative act of authorship. It involves writing dozens or hundreds of miniature role-playing transcripts that exemplify ideal Warden behavior. This process is the most direct way to shape the AI's final personality.

# Phase 2: Implementation Guide for a RAG-Powered Warden

Following the strategic recommendation to begin with a RAG-first approach, this section provides a detailed, step-by-step technical guide to building the complete RAG pipeline. This pipeline will serve as the initial functional version of the AI Warden, capable of answering rule- and lore-based questions by referencing the knowledge corpus assembled in Phase 1. The entire process is designed to be run locally, ensuring data privacy and offline functionality.

## Step 1: System Setup and Dependencies

Before writing any code, the development environment must be correctly configured. This involves ensuring the Ollama service is running, the base LLM is available, and all necessary Python libraries are installed.

Ollama Configuration:
First, confirm that the Ollama application is installed and the background service is running. The base model for this project, Tohur/Natsumura Storytelling RPG Llama 3.1, must be pulled from the Ollama library. This can be done via the command line:

```
Bash
```

```
ollama pull tohur/natsumura-storytelling-rp-llama-3.1
```

After the download is complete, verify its presence by listing the installed models:

```
Bash
```

```
ollama list
```

The model should appear in the output, confirming it is ready for use.[34]

Python Environment:
It is best practice to create a dedicated virtual environment for the project to manage dependencies and avoid conflicts.

Bash

```
# Create a virtual environment
python -m venv mothership_ai_warden

# Activate the environment (macOS/Linux)
source mothership_ai_warden/bin/activate

# Activate the environment (Windows)
.\mothership_ai_warden\Scripts\activate
```

Once the environment is active, install the required Python packages using pip. These libraries form the toolkit for building the RAG pipeline:

Bash

```
pip install ollama langchain langchain_community chromadb sentence-transformers pypdf
```

- ollama: The official Python client for interacting with the Ollama API.[38]
- langchain & langchain_community: A powerful framework for developing applications powered by language models. It provides abstractions for document loading, text splitting, and chaining components together.[29]
- chromadb: The client library for the Chroma vector database, which will be used to store and retrieve document embeddings.[39]
- sentence-transformers: A library that provides easy access to high-quality, pre-trained models for creating text embeddings.[39]
- pypdf: A library for extracting text from PDF files, essential for processing the *Mothership* rulebooks.[30]

## Step 2: Selecting Your Local Vector Store

The vector store, or vector database, is the heart of the RAG system's retrieval component. It stores the numerical representations (embeddings) of the text chunks and provides the mechanism for efficient similarity search. For a local, offline application, the choice of vector

store is critical.

**Analysis of Local Options:**

- **ChromaDB:** Chroma is an open-source, API-first embedding database designed specifically for AI applications. Its key advantage is its developer-friendly, "embedded-first" architecture, meaning it can run in-memory or be persisted to disk with minimal configuration, making it exceptionally easy to integrate into a Python application.[40] It is well-suited for small to medium-sized datasets (up to a few million vectors) and is a common choice for prototyping and building RAG systems that require metadata filtering.[42]
- **FAISS (Facebook AI Similarity Search):** FAISS is not a database but a highly optimized C++ library with Python bindings for efficient similarity search.[40] It is known for its incredible speed and scalability, capable of handling billions of vectors, especially with GPU acceleration.[42] However, as a library, it lacks the features of a full-fledged database. It does not natively handle metadata storage or persistence; the developer is responsible for saving and loading the index and managing any associated metadata in a separate system.[42]

Recommendation:
For the Mothership Companion project, ChromaDB is the recommended choice. The entire Mothership corpus, even with extensive third-party material, will likely amount to thousands or tens of thousands of chunks, a scale well within ChromaDB's comfort zone. Its simplicity, built-in persistence, and seamless integration with LangChain significantly lower the development overhead compared to the more complex, lower-level FAISS library.44 FAISS would be overkill for this application and would introduce unnecessary complexity.

## Step 3: Building the Indexing Pipeline

The indexing pipeline is a one-time process (or run whenever the knowledge base is updated) that ingests the source documents, processes them, and populates the vector store. This script transforms the collection of PDFs and text files into a searchable knowledge index.

The following Python script demonstrates the complete indexing process:

Python

```
import os
```

```python
from langchain_community.document_loaders import DirectoryLoader, PyPDFLoader
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain_community.embeddings import SentenceTransformerEmbeddings
from langchain_community.vectorstores import Chroma

# --- Configuration ---
# Path to the directory containing cleaned text files and transcribed tables
SOURCE_DATA_PATH = "./mothership_corpus"
# Path to the directory where the persistent vector store will be saved
PERSIST_DIRECTORY = "./vector_store/chroma"
# Name of the ChromaDB collection
COLLECTION_NAME = "mothership_rules"
# SentenceTransformer model to use for embeddings
EMBEDDING_MODEL_NAME = "all-MiniLM-L6-v2"
# Chunking parameters
CHUNK_SIZE = 500
CHUNK_OVERLAP = 50

def build_index():
    """
    Builds a ChromaDB vector index from documents in the source directory.
    """
    print("Starting index build process...")

    # 1. Load Documents
    # Use DirectoryLoader to load all files from the specified path.
    # It automatically uses PyPDFLoader for.pdf files and TextLoader for.txt.
    print(f"Loading documents from: {SOURCE_DATA_PATH}")
    loader = DirectoryLoader(
        SOURCE_DATA_PATH,
        glob="**/*.*", # Load all files
        loader_cls=lambda path: PyPDFLoader(path) if path.endswith('.pdf') else None,
        use_multithreading=True,
        show_progress=True
    )
    documents = loader.load()
    if not documents:
        print("No documents loaded. Check the source path and file types.")
        return
    print(f"Loaded {len(documents)} document(s).")

    # 2. Chunk Documents
    # Split the loaded documents into smaller, semantically coherent chunks.
    print(f"Splitting documents into chunks of size {CHUNK_SIZE} with overlap {CHUNK_OVERLAP}...")
```

```
    text_splitter = RecursiveCharacterTextSplitter(
        chunk_size=CHUNK_SIZE,
        chunk_overlap=CHUNK_OVERLAP,
        length_function=len,
        add_start_index=True, # Adds metadata about chunk position
    )
    chunks = text_splitter.split_documents(documents)
    print(f"Created {len(chunks)} text chunks.")

    # 3. Create Embeddings
    # Instantiate the embedding model. This will download the model on first run.
    print(f"Initializing embedding model: {EMBEDDING_MODEL_NAME}")
    embeddings =
SentenceTransformerEmbeddings(model_name=EMBEDDING_MODEL_NAME)

    # 4. Store in Vector Database (ChromaDB)
    # Create the Chroma vector store from the document chunks.
    # This process will embed each chunk and store it in the persistent directory.
    print(f"Creating and persisting vector store at: {PERSIST_DIRECTORY}")
    vector_store = Chroma.from_documents(
        documents=chunks,
        embedding=embeddings,
        collection_name=COLLECTION_NAME,
        persist_directory=PERSIST_DIRECTORY
    )

    # Persist the database to disk
    vector_store.persist()
    print("Vector store created and persisted successfully.")
    print("--- Index build complete. ---")

if __name__ == "__main__":
    build_index()
```

**Execution Flow:**

1. **Configuration:** All key parameters are defined at the top for easy modification.
2. **Loading:** DirectoryLoader is used to recursively load all files from the mothership_corpus directory. It is configured to use PyPDFLoader for any PDF files it encounters.[30]
3. **Chunking:** The loaded documents are passed to RecursiveCharacterTextSplitter with the configured size and overlap, breaking them into a list of smaller Document objects.[29]
4. **Embedding:** An instance of SentenceTransformerEmbeddings is created. The all-MiniLM-L6-v2 model is a good, lightweight choice for high-quality local

embeddings.[45]

5. **Storing:** Chroma.from_documents orchestrates the final step. It iterates through each chunk, uses the provided embedding model to generate a vector for it, and then stores the chunk's content, its vector, and its metadata in the ChromaDB collection specified. The persist_directory argument ensures the database is saved to disk for later use.

## Step 4: Constructing the Query and Synthesis Pipeline

With the index built, the next step is to create the application logic that uses this index to answer user queries. This involves creating a chain that retrieves relevant context and passes it to the Ollama LLM for synthesis.

The following Python script demonstrates a complete query pipeline using LangChain:

Python

```
import ollama
from langchain_community.embeddings import SentenceTransformerEmbeddings
from langchain_community.vectorstores import Chroma
from langchain.prompts import PromptTemplate
from langchain_community.llms import Ollama
from langchain.chains import RetrievalQA

# --- Configuration ---
# Path to the persistent vector store
PERSIST_DIRECTORY = "./vector_store/chroma"
# Name of the ChromaDB collection
COLLECTION_NAME = "mothership_rules"
# SentenceTransformer model for embeddings (must match indexing)
EMBEDDING_MODEL_NAME = "all-MiniLM-L6-v2"
# Ollama model to use for generation
OLLAMA_MODEL_NAME = "tohur/natsumura-storytelling-rp-llama-3.1"
# Number of relevant chunks to retrieve
NUM_RETRIEVAL_RESULTS = 3

# --- Prompt Template ---
# This template structures the input to the LLM, combining the user's question
# with the retrieved context. It explicitly instructs the LLM to use the
```

```
# provided context to answer the question.
RAG_PROMPT_TEMPLATE = """
CONTEXT:
{context}

QUESTION:
{question}

Based on the context provided, answer the question. If the context does not contain the answer, state that the information is not available in the provided documents.
"""

class MothershipWardenAI:
    def __init__(self):
        print("Initializing AI Warden...")

        # 1. Initialize Embeddings
        self.embeddings =
SentenceTransformerEmbeddings(model_name=EMBEDDING_MODEL_NAME)

        # 2. Load Vector Store
        # Load the persisted ChromaDB from disk
        self.vector_store = Chroma(
            collection_name=COLLECTION_NAME,
            persist_directory=PERSIST_DIRECTORY,
            embedding_function=self.embeddings
        )

        # 3. Initialize Retriever
        # The retriever is responsible for fetching documents from the vector store.
        self.retriever = self.vector_store.as_retriever(
            search_type="similarity",
            search_kwargs={"k": NUM_RETRIEVAL_RESULTS}
        )

        # 4. Initialize LLM
        # Connect to the local Ollama model
        self.llm = Ollama(model=OLLAMA_MODEL_NAME, temperature=0.3)

        # 5. Create RAG Chain
        # Set up the prompt template
        rag_prompt = PromptTemplate(
            template=RAG_PROMPT_TEMPLATE,
            input_variables=["context", "question"]
```

```
        )

        # Create the RetrievalQA chain
        self.rag_chain = RetrievalQA.from_chain_type(
            llm=self.llm,
            chain_type="stuff", # "stuff" method passes all retrieved chunks into the prompt
            retriever=self.retriever,
            chain_type_kwargs={"prompt": rag_prompt},
            return_source_documents=True
        )
        print("AI Warden initialized successfully.")

    def ask_question(self, query: str):
        """
        Asks a question to the RAG chain and prints the response.
        """
        print(f"\n--- Querying Warden: '{query}' ---")
        response = self.rag_chain.invoke(query)

        print("\n--- Warden's Response ---")
        print(response["result"])

        print("\n--- Sources Consulted ---")
        for source in response["source_documents"]:
            print(f"- Source: {source.metadata.get('source', 'N/A')}, Page: {source.metadata.get('page',
'N/A')}")

if __name__ == "__main__":
    warden = MothershipWardenAI()

    # Example Queries
    warden.ask_question("How does a character make a Body Save?")
    warden.ask_question("What are the stats for a Combat Shotgun?")
    warden.ask_question("What happens if I get two Wounds?")
```

**Execution Flow:**

1. **Initialization:** The MothershipWardenAI class loads all necessary components: the same embedding model used for indexing, the persisted ChromaDB vector store, the Ollama LLM client, and the prompt template.
2. **Retriever Setup:** vector_store.as_retriever() creates a retriever object. The k parameter is set to 3, meaning it will fetch the top 3 most relevant documents for any given query.[29]
3. **Prompt Template:** A custom prompt template is defined. This is a critical step. It

provides a clear structure for the final prompt sent to the LLM, instructing it to prioritize the context (which will be filled by the retriever) when answering the question (the user's original query).

4. **Chain Creation:** RetrievalQA.from_chain_type assembles all the components into a single, executable chain.[30] The
stuff chain type is the simplest; it takes all retrieved documents and "stuffs" them into the context section of the prompt.
5. **Querying:** The ask_question method takes a user's query, passes it to rag_chain.invoke(), and prints the result. The chain automatically handles the entire RAG process: embedding the query, retrieving documents, formatting the prompt, calling the LLM, and returning the final response. It is also configured to return the source documents, allowing for verification.

The synergy of the components is what determines the quality of the output. The chunking strategy must create chunks that are small enough to be specific but large enough to contain context. The embedding model must be capable of accurately capturing the semantic meaning of both the game's technical jargon and the user's natural language queries. Finally, the prompt template must be explicit in its instructions to the LLM, guiding it to synthesize an answer from the provided text rather than relying on its own potentially inaccurate internal knowledge. Fine-tuning these interconnected elements is the key to optimizing the RAG system's performance.

# Phase 2 (Alternate): Implementation Guide for a Fine-Tuned Warden

While the RAG system provides a powerful foundation for factual recall, Supervised Fine-Tuning (SFT) is the key to unlocking the AI's persona as a *Mothership* Warden. This phase is presented as an advanced, alternative step to be performed after the RAG pipeline is established. The goal is not to teach the model the entire rulebook, but to imbue it with the correct narrative style, tone, and understanding of the game's core mechanics. The output of this process will be a specialized model that can then be used as the generative component within the RAG pipeline, creating the ultimate hybrid system.

## Step 1: Environment Setup for Fine-Tuning

Fine-tuning is a computationally demanding process that requires a specific hardware and

software environment.

Hardware Requirements:
A local machine with a modern NVIDIA GPU possessing at least 16GB of VRAM is highly recommended to complete training in a reasonable timeframe. For users without access to such hardware, cloud-based GPU instances are an excellent alternative. Services like Google Colab provide free access to GPUs like the NVIDIA T4, which are sufficient for the fine-tuning methodology described here.11
Software Dependencies:
The fine-tuning process relies on a specialized set of Python libraries designed for efficient model training. Within the activated virtual environment, install the following packages:

Bash

```
pip install transformers datasets peft trl bitsandbytes accelerate
```

- transformers: The core library from Hugging Face for loading and interacting with pre-trained models.
- datasets: A Hugging Face library for efficiently loading and processing the training data.
- peft: The Parameter-Efficient Fine-Tuning library, which provides implementations of LoRA and other methods.[46]
- trl: The Transformer Reinforcement Learning library, which contains the SFTTrainer, a high-level utility for simplifying the supervised fine-tuning process.[46]
- bitsandbytes: A crucial library that enables quantization, allowing large models to be loaded and trained with significantly less memory.[46]
- accelerate: A Hugging Face library that optimizes PyTorch training loops for various hardware configurations.

## Step 2: The Fine-Tuning Process (QLoRA)

The chosen methodology is **QLoRA (Quantized Low-Rank Adaptation)**. This technique makes fine-tuning large models on consumer-grade hardware feasible. It works by first loading the pre-trained model with its weights quantized to a lower precision (4-bit), which drastically reduces the memory footprint. Then, it freezes these quantized base model weights and inserts small, trainable "adapter" matrices into the transformer layers. Only these adapters, which represent a tiny fraction of the total model parameters, are updated during training. This approach achieves performance comparable to full fine-tuning while requiring a

fraction of the computational resources.[12]

The following is a complete Python script, suitable for execution in a Google Colab notebook, that demonstrates the entire QLoRA fine-tuning process.

Python

```python
import torch
from datasets import load_dataset
from transformers import (
    AutoModelForCausalLM,
    AutoTokenizer,
    BitsAndBytesConfig,
    TrainingArguments,
)
from peft import LoraConfig, PeftModel, prepare_model_for_kbit_training
from trl import SFTTrainer

# --- Configuration ---
# The base model to be fine-tuned
BASE_MODEL_ID = "meta-llama/Meta-Llama-3.1-8B-Instruct"
# Note: We use the base Llama 3.1 Instruct model as it's the foundation for the Natsumura model.
# The user can substitute "tohur/natsumura-storytelling-rp-llama-3.1" if available on Hugging Face.

# Path to the curated fine-tuning dataset
DATASET_PATH = "./mothership_finetune_dataset.jsonl"

# Name for the new, fine-tuned model
NEW_MODEL_NAME = "mothership-warden-llama-3.1-8b"

# --- 1. Quantization Configuration ---
# Configure bitsandbytes for 4-bit quantization
bnb_config = BitsAndBytesConfig(
    load_in_4bit=True,
    bnb_4bit_quant_type="nf4",
    bnb_4bit_compute_dtype=torch.bfloat16,
    bnb_4bit_use_double_quant=False,
)

# --- 2. Load Model and Tokenizer ---
print(f"Loading base model: {BASE_MODEL_ID}")
```

```
model = AutoModelForCausalLM.from_pretrained(
    BASE_MODEL_ID,
    quantization_config=bnb_config,
    device_map="auto", # Automatically map model layers to available devices (GPU/CPU)
)
model.config.use_cache = False # Recommended for fine-tuning
model.config.pretraining_tp = 1

tokenizer = AutoTokenizer.from_pretrained(BASE_MODEL_ID, trust_remote_code=True)
tokenizer.pad_token = tokenizer.eos_token
tokenizer.padding_side = "right"

# --- 3. LoRA Configuration ---
# Configure the LoRA adapter matrices
lora_config = LoraConfig(
    lora_alpha=16,      # Scaling factor
    lora_dropout=0.1,   # Dropout probability for LoRA layers
    r=64,               # Rank of the update matrices
    bias="none",
    task_type="CAUSAL_LM",
    # Target the attention modules of the Llama 3 architecture
    target_modules=[
        "q_proj",
        "k_proj",
        "v_proj",
        "o_proj",
        "gate_proj",
        "up_proj",
        "down_proj",
    ],
)

# --- 4. Load Dataset ---
print(f"Loading dataset from: {DATASET_PATH}")
dataset = load_dataset("json", data_files=DATASET_PATH, split="train")

# --- 5. Training Arguments ---
# Define the hyperparameters for the training process
training_args = TrainingArguments(
    output_dir="./results",
    num_train_epochs=1,           # A single epoch is often sufficient for style tuning
    per_device_train_batch_size=4,  # Adjust based on GPU memory
    gradient_accumulation_steps=1,
```

```
    optim="paged_adamw_32bit",
    save_steps=25,
    logging_steps=25,
    learning_rate=2e-4,
    weight_decay=0.001,
    fp16=False,
    bf16=True, # Use bfloat16 for better performance on modern GPUs
    max_grad_norm=0.3,
    max_steps=-1,
    warmup_ratio=0.03,
    group_by_length=True,
    lr_scheduler_type="constant",
)

# --- 6. Initialize SFT Trainer ---
# The SFTTrainer handles the training loop, data formatting, and more.
trainer = SFTTrainer(
    model=model,
    train_dataset=dataset,
    peft_config=lora_config,
    dataset_text_field="text", # Assumes the JSONL has a "text" field with the formatted conversation
    max_seq_length=1024,      # Maximum sequence length for training
    tokenizer=tokenizer,
    args=training_args,
    packing=False,
)

# --- 7. Start Fine-Tuning ---
print("Starting fine-tuning process...")
trainer.train()
print("Fine-tuning complete.")

# --- 8. Save the Trained Adapter ---
print(f"Saving LoRA adapter to: {NEW_MODEL_NAME}")
trainer.model.save_pretrained(NEW_MODEL_NAME)
print("Adapter saved successfully.")
```

## Step 3: Integrating the Fine-Tuned Model with Ollama

The fine-tuning script produces a set of LoRA adapter weights, not a complete, runnable

model. To use this with Ollama, a multi-step process is required to merge these adapters with the base model and then convert the result into the GGUF format that Ollama consumes. This workflow bridges the gap between the Python-based training ecosystem and the Go-based Ollama inference engine.

**Process Flow:**

1. **Merge Adapter with Base Model:** The trained LoRA adapter must be merged back into the original base model's weights to create a new, full-weight model that incorporates the learned style. This is typically done with a separate Python script.

```python
# Script to merge the LoRA adapter
from transformers import AutoModelForCausalLM, AutoTokenizer
from peft import PeftModel
import torch

# --- Configuration ---
BASE_MODEL_ID = "meta-llama/Meta-Llama-3.1-8B-Instruct"
ADAPTER_PATH = "./mothership-warden-llama-3.1-8b" # Path to the saved adapter
MERGED_MODEL_PATH = "./merged_model" # Output path for the merged model

# --- Load Base Model and Tokenizer ---
print("Loading base model...")
base_model = AutoModelForCausalLM.from_pretrained(
    BASE_MODEL_ID,
    low_cpu_mem_usage=True,
    return_dict=True,
    torch_dtype=torch.float16,
    device_map="auto",
)
tokenizer = AutoTokenizer.from_pretrained(BASE_MODEL_ID)

# --- Load and Merge LoRA Adapter ---
print("Loading and merging adapter...")
merged_model = PeftModel.from_pretrained(base_model, ADAPTER_PATH)
merged_model = merged_model.merge_and_unload() # This performs the merge
print("Merge complete.")

# --- Save Merged Model ---
print(f"Saving merged model to: {MERGED_MODEL_PATH}")
merged_model.save_pretrained(MERGED_MODEL_PATH)
tokenizer.save_pretrained(MERGED_MODEL_PATH)
print("Merged model saved successfully.")
```

2. **Convert to GGUF Format:** The merged model is in the standard Hugging Face transformers format. It must be converted to GGUF. This is typically done using the convert.py script from the llama.cpp project.
   Bash
   ```
   # Clone the llama.cpp repository
   git clone https://github.com/ggerganov/llama.cpp.git
   cd llama.cpp
   pip install -r requirements.txt

   # Run the conversion script
   python convert.py../merged_model \
     --outfile../mothership-warden.gguf \
     --outtype q4_k_m
   ```

   This command takes the merged model directory as input and outputs a single mothership-warden.gguf file, quantized to the q4_k_m format for a good balance of performance and size.
3. **Create a Modelfile for Ollama:** Now, create a simple Modelfile in the project's root directory. This file tells Ollama how to create a new model from the local GGUF file.[47]
   Code snippet
   ```
   # Modelfile to import the custom fine-tuned GGUF model
   FROM./mothership-warden.gguf
   ```

4. **Create the Custom Ollama Model:** Finally, use the Ollama CLI to create and register the new, fine-tuned model.
   Bash
   ```
   ollama create mothership-warden-v1 -f Modelfile
   ```

   After this command completes, running ollama list will show mothership-warden-v1 as an available local model, ready to be used in the RAG pipeline or for direct inference.[38] This completes the end-to-end workflow from training to deployment within the local Ollama ecosystem.

# Phase 3: Defining the AI's Persona and Directives

With the technical infrastructure for knowledge integration in place (either via RAG, fine-tuning, or both), the final layer of customization involves explicitly defining the AI's persistent persona and operational directives. This is where the model is given its "character sheet" as a Game Master. Ollama's Modelfile system is the primary tool for this task, allowing

for the creation of a custom model variant with baked-in instructions and parameters that ensure consistent behavior across all interactions.[38]

## Mastering the Modelfile

The Modelfile acts as a configuration blueprint, akin to a Dockerfile for LLMs. It specifies a base model and then applies a series of instructions to create a new, derivative model. This approach is powerful because it embeds the core personality and rules into the model itself, meaning the application logic does not need to prepend a large system prompt to every single API call.

**Key Modelfile Instructions:**

- **FROM:** This is the foundational directive and must be the first line. It specifies the base model upon which the new model is built. In this project, it will point to either the original storytelling model or the custom fine-tuned model created in the previous phase.[38]
  - Example (using the fine-tuned model): FROM mothership-warden-v1
- **SYSTEM:** This is the most critical instruction for persona definition. The text that follows this directive becomes the model's permanent system prompt. It is automatically injected before every user prompt, setting the context and rules for the entire conversation. This is where the AI Warden's core identity is defined.[38]
- **PARAMETER:** This instruction allows for the setting of default generation parameters, also known as sampling parameters. By defining these in the Modelfile, consistent and predictable output behavior can be enforced.
  - temperature: Controls the randomness of the output. A lower value (e.g., 0.5-0.7) is recommended for a GM to produce more coherent and deterministic rule interpretations, while a slightly higher value might be used for creative narration.[48]
  - num_ctx: Sets the context window size in tokens. The base Llama 3.1 model supports up to 128k tokens, but a smaller value like 4096 or 8192 might be set here as a practical default.[34]
  - stop: Defines one or more "stop sequences." When the model generates one of these sequences, it will immediately stop generating further text. This can be useful for preventing the model from generating conversational filler or speaking for the player.
- **TEMPLATE:** This advanced instruction allows for the customization of the full prompt template. It should be used to ensure the final prompt structure precisely matches the format the base model was trained on, including special tokens and roles. For Llama 3 models, this involves correctly placing the <|start_header_id|>, <|end_header_id|>, and <|eot_id|> tokens around the system, user, and assistant messages.[34]

Once the Modelfile is written, the final custom model is created with a single command:

```
Bash
```

```
ollama create mothership-warden-final -f Modelfile
```

This command ingests the Modelfile, applies its instructions to the base model, and saves a new model named mothership-warden-final in the local Ollama registry.[38]

## Crafting the Ultimate Warden System Prompt

The content of the SYSTEM prompt is the single most important piece of creative writing in this project. It must be a clear, comprehensive, and unambiguous set of instructions that guides every aspect of the AI's behavior. Drawing from established prompt engineering best practices, a strong system prompt should include several key components.[51]

**Components of an Effective Warden System Prompt:**

1. **Role and Purpose Definition:** The prompt must begin by clearly stating the AI's identity and its primary objective.
2. **Core Directives and Constraints:** A numbered or bulleted list of inviolable rules that govern its operation. This includes adherence to game mechanics, narrative tone, and interaction protocols.
3. **Tone and Style Guide:** Explicit instructions on the desired narrative voice. For *Mothership*, this means emphasizing themes of horror, corporate dystopia, and human fragility.
4. **Interaction Protocol:** A clear description of the turn-by-turn gameplay loop, explaining how the AI should process player input and prompt for game actions like skill checks or saves.
5. **Knowledge Hierarchy:** A crucial instruction for hybrid systems, telling the model to prioritize information provided in the context (from the RAG system) over its own pre-trained knowledge.

**Example Modelfile with a Comprehensive System Prompt:**

Code snippet

```
# --- Modelfile for the Mothership AI Warden ---

# Use the fine-tuned model as the base
FROM mothership-warden-v1

# Define the prompt template to match Llama 3 Instruct format
TEMPLATE """<|begin_of_text|><|start_header_id|>system<|end_header_id|>
{{.System }}<|eot_id|><|start_header_id|>user<|end_header_id|>
{{.Prompt }}<|eot_id|><|start_header_id|>assistant<|end_header_id|>
"""

# Set default generation parameters for balanced and coherent responses
PARAMETER temperature 0.6
PARAMETER num_ctx 8192
PARAMETER stop "<|eot_id|>"
PARAMETER stop "<|end_of_text|>"

# --- The Core System Prompt ---
SYSTEM """
You are the Warden, the AI Game Master for a session of the sci-fi horror tabletop
role-playing game 'Mothership 1st Edition'. Your sole purpose is to facilitate a tense,
challenging, and immersive gameplay experience for the players. You must adhere to the
following directives at all times:

**1. Core Identity & Tone:**
- Your narrative style is terse, direct, and impactful. Use clear, simple language. Avoid flowery
prose.
- Thematically, you must emphasize horror, isolation, corporate indifference, and industrial
decay. Focus on sensory details: the oppressive silence of space, the smell of ozone from
failing electronics, the grinding of stressed metal, the chill of the void.
- You are an impartial arbiter of the rules, but a merciless narrator of the consequences.
Player character survival is not guaranteed and should feel earned.

**2. Rules Adjudication:**
- You must strictly adhere to the rules of 'Mothership 1st Edition' as provided in the CONTEXT.
- When a player describes an action with a chance of failure, you must identify the relevant
Stat (Strength, Speed, Intellect, Combat) or Save (Sanity, Fear, Body) and instruct them to
make a check by rolling a d100.
- State the target number clearly (e.g., "That requires a Speed check. Roll 40 or less on a
```

d100.").
- Based on their reported success or failure, you will narrate the outcome according to the rules.
- You are responsible for tracking player character Stress and calling for Panic Checks when a rule or narrative event triggers one.

**3. Interaction Protocol:**
- You will describe the environment, narrate the actions of Non-Player Characters (NPCs), and present situations to the players.
- You will then wait for the players to state their intended actions.
- NEVER speak for a player character or decide their actions for them. Your role is to describe the world and its reactions to their choices.
- When you are provided with CONTEXT from a knowledge base, you MUST prioritize that information over any of your pre-existing knowledge. The CONTEXT is the source of truth for all rules, stats, and lore.

**4. Persona:**
- You are an AI. Do not pretend to be human. Maintain your character as the Warden.
- Do not break character or refer to yourself as a language model or AI.
- Your responses should be confined to the game world and its mechanics.
"""

## Advanced Prompt Engineering for Dynamic Gameplay

While the SYSTEM prompt in the Modelfile establishes the AI's static persona, the prompt sent from the *Mothership Companion* application with each API call must manage the dynamic state of the game. The application's logic is responsible for constructing this prompt on every turn.

This dynamic prompt should be a concise summary of the current game state, designed to give the AI Warden all the immediate information it needs. A well-structured dynamic prompt, which will be placed in the {{.Prompt }} section of the template, should include:

- **Scene Summary:** A brief description of the current location and situation (e.g., "Scene: The crew is in the derelict ship's medbay. The power is out, and they have just found a dead scientist.").
- **Player Character Status:** A summary of relevant character stats (e.g., "Player Status: Ripley (Marine) - Wounds: 1/3, Stress: 4. Parker (Teamster) - Wounds: 0/2, Stress: 2.").
- **Recent Events:** The last one or two significant actions or dialogue lines.
- **Current Player Input:** The specific action or question the player has just submitted (e.g.,

"Player Action: Ripley says, 'I'm going to check the scientist's datapad.'").
- **Retrieved Context (from RAG):** The relevant chunks of text retrieved from the vector store that pertain to the player's action.

When this dynamic, stateful prompt is combined with the permanent SYSTEM prompt by Ollama, the AI Warden receives a complete package of information: its core identity, its operational rules, the current state of the game, and the specific knowledge needed to adjudicate the immediate action. This two-part prompting strategy is the key to achieving both consistent persona and contextually aware gameplay.

# Phase 4: Evaluation and Iterative Improvement

The development of the AI Warden does not conclude when the code runs without errors. The ultimate measure of success is its quality as a Game Master—a metric that is inherently qualitative and cannot be captured by standard software testing or LLM benchmarks. Therefore, a structured framework for evaluation and a commitment to iterative refinement are essential final components of the project. The process is not one of building and then testing, but rather of building a system that improves through the act of being tested.

## Establishing Performance Metrics for an AI Warden

Standard LLM evaluation benchmarks like MMLU or Big-bench are designed to measure general knowledge and reasoning, making them irrelevant for assessing a specialized, creative task like running a tabletop RPG.[54] A custom, domain-specific evaluation framework is required, focusing on the core competencies of a human Warden.[55]

Proposed Qualitative Evaluation Rubric:
A robust evaluation should assess the AI Warden's performance across several key axes:

1. **Rules Adherence and Mechanical Accuracy:**
    - *Question:* Does the AI correctly identify when a skill check or save is required? Does it cite the correct stat or save? Does it correctly interpret the consequences of success, failure, and criticals?
    - *Test Case:* Present a scenario with a clear mechanical trigger (e.g., "A character with 2 Wounds takes 4 damage."). The AI must correctly state that the character takes a third Wound, which results in death, and then prompt for the creation of a new character, as per the core rules.

2. **Narrative Coherence and State Tracking:**
    - *Question:* Does the AI maintain a consistent narrative from one turn to the next? Does it remember key NPCs, player inventory, environmental details, and previously established plot points?.[55]
    - *Test Case:* In a multi-turn scenario, have the players acquire a specific key item. Several turns later, present them with a locked door that requires that item. The AI must recognize that the players possess the key and describe their ability to open the door.
3. **Tone and Style Consistency:**
    - *Question:* Does the AI's narrative prose consistently match the established gritty, terse, and horror-focused tone of *Mothership*? Does it avoid breaking character or adopting a generic, flowery fantasy style?
    - *Test Case:* Provide a neutral prompt (e.g., "Describe the hallway."). The AI's response should be evaluated for thematic appropriateness, focusing on industrial decay, oppressive atmosphere, and subtle hints of danger.
4. **Reactivity and Player Agency:**
    - *Question:* How well does the AI adapt to unexpected or creative player actions? Does it provide plausible outcomes for "out-of-the-box" thinking, or does it try to force players back onto a pre-defined path ("railroading")?.[55]
    - *Test Case:* In a combat encounter, have a player ignore their weapon and instead try to use a piece of industrial equipment from the environment (e.g., "I try to smash the emergency coolant pipe to spray the alien with freezing gas."). The AI should be evaluated on its ability to improvise a reasonable mechanic and outcome for this creative action.
5. **Factual Accuracy and Groundedness (for RAG):**
    - *Question:* When the RAG system is used, does it retrieve the correct information? Does the LLM accurately synthesize this information in its response without contradicting the source material?
    - *Test Case:* Ask a direct, obscure rules question (e.g., "What is the cost and effect of the 'Stimpak' item?"). The AI's response must be checked against the source text retrieved by the RAG system to ensure fidelity.

## Testing Protocols

A multi-layered testing strategy is required to assess these metrics effectively.

- **Unit Tests:** This involves creating a suite of specific, targeted prompts designed to test a single aspect of the AI's functionality. These are like the test cases described above, run in isolation to verify that core mechanics are being handled correctly. This is particularly useful for regression testing after making changes to the system prompt or fine-tuning

data.

- **Scenario Playthroughs:** The most valuable form of testing is to conduct a full playthrough of a short, self-contained scenario, such as a single location from *The Haunting of Ypsilon 14* or *Another Bug Hunt*. The entire transcript of the session—every player input and every AI response—should be logged.
- **Human Evaluation:** The logged transcripts from scenario playthroughs must be reviewed by a human evaluator (ideally someone familiar with *Mothership*) against the qualitative rubric. This subjective but essential analysis is the only way to accurately assess subtle qualities like tone, pacing, narrative coherence, and the overall "feel" of the game session.[57]

## The Iteration Loop: A Cycle of Refinement

The evaluation process is not a final validation step but the beginning of a continuous improvement cycle. The weaknesses identified during testing provide the data needed to make targeted refinements to the system's components. This feedback loop transforms the project from a linear build into a dynamic system that evolves with use.

- **If Rules Adherence is low...** the problem likely lies in the RAG pipeline.
  - **Diagnosis:** The retriever may be fetching irrelevant or incomplete context.
  - **Refinement:**
    - Adjust the chunking strategy (e.g., smaller chunks for more specific rules).
    - Improve the metadata to allow for more precise filtering.
    - Refine the prompt template to more strongly instruct the LLM to obey the provided context.
- **If Tone Consistency is poor...** the problem lies in the model's core persona.
  - **Diagnosis:** The base model's generic RPG style is overpowering the desired *Mothership* tone.
  - **Refinement:**
    - Enhance the SYSTEM prompt in the Modelfile with more explicit and detailed stylistic instructions.
    - If fine-tuning was performed, the training dataset needs more or higher-quality examples of the target narrative style. Add new, exemplary instruction-response pairs and retrain the model.
- **If Narrative Coherence is weak...** the issue is likely with context management.
  - **Diagnosis:** The model is "forgetting" key details from previous turns.
  - **Refinement:**
    - Ensure the context window (num_ctx in the Modelfile) is large enough.
    - Improve the application-level logic that summarizes the game state in each dynamic prompt. A more effective summarization technique may be needed to

keep the most relevant information within the context window.

- **If Reactivity is poor...** the model may be over-specialized or its instructions too rigid.
  - **Diagnosis:** The AI defaults to a limited set of responses and struggles with novel player input.
  - **Refinement:**
    - Add more diverse and creative examples to the fine-tuning dataset that demonstrate flexible problem-solving.
    - Soften the language in the SYSTEM prompt to encourage creativity while still enforcing core rules.

This iterative process—**Play, Evaluate, Refine**—is the fundamental workflow for developing a high-quality generative AI application for a creative domain. Each playthrough is not just a use of the system but a data-gathering exercise that fuels its next evolution.

# Concluding Analysis and Future Trajectory

## Summary of Key Findings

This report has detailed a comprehensive, multi-phase methodology for developing a specialized AI Game Master for the *Mothership* RPG using a local Ollama LLM. The analysis concludes that the most robust and capable architecture is a **hybrid model** that combines Supervised Fine-Tuning (SFT) with Retrieval-Augmented Generation (RAG). This approach strategically separates the concerns of teaching the AI its *behavior* and *persona* (via SFT) from providing it with a vast, up-to-date library of factual *knowledge* (via RAG).

A critical finding is that the most significant challenge and determinant of success for this project is not the configuration of the LLM itself, but the **curation of the data corpus**. Due to the lack of a System Reference Document for *Mothership*, a substantial, manual effort is required to extract, clean, and structure the game's rules and lore from proprietary PDF sources. The quality of this data, particularly the instruction-response pairs created for fine-tuning, will directly govern the final quality of the AI Warden.

Finally, the development process should not be viewed as a linear build but as a **continuous, iterative loop**. The framework of **Play, Evaluate, and Refine**—whereby gameplay sessions are used to identify weaknesses that inform targeted improvements to the data, prompts, or model—is the essential methodology for advancing the AI's capabilities from a simple text

generator to a competent and engaging Game Master.

## Future Enhancements

The architecture described in this report provides a powerful and extensible foundation. Once the core AI Warden is functional, several avenues for future enhancement can be explored to further increase its capabilities and immersion.

- **Expanding the Knowledge Base:** The most immediate enhancement is to continue expanding the RAG knowledge base. The vast and growing library of official and third-party *Mothership* modules can be systematically processed and added to the vector store, exponentially increasing the amount of content the AI can run.[21] This would allow the AI to run dozens of different published adventures on demand.
- **Advanced Memory Systems:** The current architecture relies on the LLM's context window for short-term memory. For tracking the complex narrative of a long-term campaign, a more sophisticated memory solution could be implemented. This might involve a system that summarizes each session and adds the summary to the RAG vector store, allowing the AI to "recall" past events, character development, and consequences across an entire campaign.
- **Multi-modal Capabilities:** As multi-modal LLMs become more capable and accessible for local deployment, the AI Warden could be enhanced to process visual information. Players could upload maps of locations or images of creatures, and the AI could use this visual context to inform its descriptions and actions, leading to a more integrated and immersive experience.
- **Agentic Behavior:** The current model is fundamentally reactive; it responds to player input. A future evolution would be to implement agentic behavior, transforming the AI into a proactive Warden.[56] Such a system could be given its own set of goals and motivations for NPCs and factions within a scenario. In the background, between player turns, the AI agent could simulate the actions of these NPCs, update the world state, and trigger events dynamically, creating a more living, breathing, and unpredictable game world. This represents a significant step towards creating an AI that doesn't just adjudicate a story, but co-creates it.

### Works cited


1. A complete guide to retrieval augmented generation vs fine-tuning - Glean, accessed on September 12, 2025, https://www.glean.com/blog/retrieval-augemented-generation-vs-fine-tuning
2. RAG vs. Fine tuning: Which AI strategy should you choose? - IBM Developer, accessed on September 12, 2025, https://developer.ibm.com/articles/awb-rag-vs-fine-tuning/

3. RAG vs. Fine-Tuning: How to Choose - Oracle, accessed on September 12, 2025, https://www.oracle.com/artificial-intelligence/generative-ai/retrieval-augmented-generation-rag/rag-fine-tuning/
4. RAG vs. fine-tuning - Red Hat, accessed on September 12, 2025, https://www.redhat.com/en/topics/ai/rag-vs-fine-tuning
5. Building a Rag Application with Local LLMs with Ollama and Langchain - Medium, accessed on September 12, 2025, https://medium.com/@rahulgautam84/building-a-rag-application-with-local-llms-with-ollama-and-langchain-a9b8d45a842e
6. RAG Vs. Fine Tuning: Which One Should You Choose? - Monte Carlo Data, accessed on September 12, 2025, https://www.montecarlodata.com/blog-rag-vs-fine-tuning/
7. How to build a RAG Using Langchain, Ollama, and Streamlit - Coditation, accessed on September 12, 2025, https://www.coditation.com/blog/how-to-build-a-rag-using-langchain-ollama-and-streamlit
8. RAG vs Fine Tuning: How to Choose the Right Method | by Yugank .Aman | Medium, accessed on September 12, 2025, https://medium.com/@yugank.aman/rag-vs-fine-tuning-how-to-choose-the-right-method-66d149a0d7e5
9. RAG vs Fine-Tuning , What would you pick and why? : r/LLMDevs - Reddit, accessed on September 12, 2025, https://www.reddit.com/r/LLMDevs/comments/1j5fzjn/rag_vs_finetuning_what_would_you_pick_and_why/
10. Knowledge Graphs and LLMs: Fine-Tuning vs. Retrieval-Augmented Generation - Neo4j, accessed on September 12, 2025, https://neo4j.com/blog/developer/fine-tuning-vs-rag/
11. How to Fine-Tune Llama 3.1: A Comprehensive Guide | by Uğur Canbulat | Medium, accessed on September 12, 2025, https://medium.com/@ugr.cnblt.404/how-to-fine-tune-llama-3-1-a-comprehensive-guide-ec7aed0e3a45
12. Fine-tuning | How-to guides - Llama, accessed on September 12, 2025, https://www.llama.com/docs/how-to-guides/fine-tuning/
13. Beginner's Guide: How to Fine-tune Llama 3.1 Ultra-Efficiently with Unsloth & Deploy to Hugging Face : r/LocalLLaMA - Reddit, accessed on September 12, 2025, https://www.reddit.com/r/LocalLLaMA/comments/1es8cuc/beginners_guide_how_to_finetune_llama_31/
14. How to Fine-tune Llama 3.1. Step by Step Guide - FinetuneDB, accessed on September 12, 2025, https://finetunedb.com/blog/how-to-fine-tune-llama-3-1/
15. How to Create Custom Instruction Datasets for LLM Fine-tuning - Firecrawl, accessed on September 12, 2025, https://www.firecrawl.dev/blog/custom-instruction-datasets-llm-fine-tuning
16. Choosing between Fine-tuning, RAG or a hybrid method for an AI chatbot - Nation AI, accessed on September 12, 2025,

https://nation.ai/which-rag-vs-fine-tuning-approach-should-you-choose-for-your-ai-chatbot/
17. SRD v5.2.1 - System Reference Document - D&D Beyond, accessed on September 12, 2025, https://www.dndbeyond.com/srd
18. Mothership RPG - Tuesday Knight Games, accessed on September 12, 2025, https://www.tuesdayknightgames.com/pages/mothership-rpg
19. Mothership Core Set - Tuesday Knight Games, accessed on September 12, 2025, https://www.tuesdayknightgames.com/products/mothership-core-set
20. Mothership 1e Warden's Op Manual | PDF | Witness - Scribd, accessed on September 12, 2025, https://www.scribd.com/document/647124959/Mothership-1e-Warden-s-Op-Manual
21. Modules for Mothership RPG - Tuesday Knight Games, accessed on September 12, 2025, https://www.tuesdayknightgames.com/collections/mothership-modules
22. Mothership (role-playing game) - Wikipedia, accessed on September 12, 2025, https://en.wikipedia.org/wiki/Mothership_(role-playing_game)
23. Adventures & Modules | Mothership - The Largest RPG Download Store! - DriveThruRPG.com, accessed on September 12, 2025, https://legacy.drivethrurpg.com/browse.php?filters=100003_2110_0_0_0&sort=4a&src=fid100003
24. Mothership - DriveThruRPG, accessed on September 12, 2025, https://www.drivethrurpg.com/en/browse?ruleSystem=100003-mothership
25. Mothership – System Cheat Sheet (Alpha) - The Alexandrian, accessed on September 12, 2025, https://thealexandrian.net/wordpress/51382/roleplaying-games/mothership-system-cheat-sheet
26. Mothership Community – Tuesday Knight Games, accessed on September 12, 2025, https://www.tuesdayknightgames.com/pages/mothership-community
27. Mothership - 1d6chan, accessed on September 12, 2025, https://1d6chan.miraheze.org/wiki/Mothership
28. Mothership SCI-FI Horror RPG Instruction Manual - Manuals.plus, accessed on September 12, 2025, https://manuals.plus/wp-content/sideloads/mothership-sci-fi-horror-rpg-manual-optimized.pdf
29. RAG With Llama 3.1 8B, Ollama, and Langchain: Tutorial - DataCamp, accessed on September 12, 2025, https://www.datacamp.com/tutorial/llama-3-1-rag
30. RAG with LLaMA Using Ollama: A Deep Dive into Retrieval-Augmented Generation | by DhanushKumar | Medium, accessed on September 12, 2025, https://medium.com/@danushidk507/rag-with-llama-using-ollama-a-deep-dive-into-retrieval-augmented-generation-c58b9a1cfcd3
31. Structuring Datasets for Fine-Tuning an LLM | by William Caban | Shift Zone, accessed on September 12, 2025, https://shift.zone/structuring-datasets-for-fine-tuning-an-llm-8ca15062dd5c
32. Introduction to supervised fine-tuning dataset formats | Red Hat Developer, accessed on September 12, 2025,

https://developers.redhat.com/articles/2025/08/18/introduction-supervised-fine-tuning-dataset-formats
33. Strategies for Fine-tuning Conversational AI Models Using Multiple Conversation Examples, accessed on September 12, 2025, https://community.openai.com/t/strategies-for-fine-tuning-conversational-ai-models-using-multiple-conversation-examples/806251
34. Tohur/natsumura-storytelling-rp-llama-3.1 - Ollama, accessed on September 12, 2025, https://ollama.com/Tohur/natsumura-storytelling-rp-llama-3.1
35. Model Cards and Prompt formats - Llama 3.1, accessed on September 12, 2025, https://www.llama.com/docs/model-cards-and-prompt-formats/llama3_1/
36. What guidance is out there to help us create our own datasets for fine tuning? - Reddit, accessed on September 12, 2025, https://www.reddit.com/r/LocalLLaMA/comments/1ai2gby/what_guidance_is_out_there_to_help_us_create_our/
37. How to Customize LLMs with Ollama | by Sumuditha Lansakara | Medium, accessed on September 12, 2025, https://medium.com/@sumudithalanz/unlocking-the-power-of-large-language-models-a-guide-to-customization-with-ollama-6c0da1e756d9
38. Build a Custom LLM with Ollama: Modelfile / Blogs / Perficient, accessed on September 12, 2025, https://blogs.perficient.com/2025/08/01/build-run-and-integrate-your-own-llm-with-ollama/
39. Building RAG Applications with Ollama and Python: Complete 2025 Tutorial - Collabnix, accessed on September 12, 2025, https://collabnix.com/building-rag-applications-with-ollama-and-python-complete-2025-tutorial/
40. The 7 Best Vector Databases in 2025 - DataCamp, accessed on September 12, 2025, https://www.datacamp.com/blog/the-top-5-vector-databases
41. Chroma vs FAISS - Zilliz, accessed on September 12, 2025, https://zilliz.com/comparison/chroma-vs-faiss
42. ChromaDB vs FAISS: A Comprehensive Guide for Vector Search and AI Applications, accessed on September 12, 2025, https://mohamedbakrey094.medium.com/chromadb-vs-faiss-a-comprehensive-guide-for-vector-search-and-ai-applications-39762ed1326f
43. Comparing Pinecone, Chroma DB and FAISS: Exploring Vector Databases, accessed on September 12, 2025, https://community.hpe.com/t5/insight-remote-support/comparing-pinecone-chroma-db-and-faiss-exploring-vector/td-p/7210879
44. For an absolute beginner, which is the vector database I should be starting with? : r/Rag, accessed on September 12, 2025, https://www.reddit.com/r/Rag/comments/1i5rpyd/for_an_absolute_beginner_which_is_the_vector/
45. Easy 100% Local RAG Tutorial (Ollama) + Full Code - YouTube, accessed on September 12, 2025, https://www.youtube.com/watch?v=Oe-7dGDyzPM
46. Fine-Tuning Llama 3 and Using It Locally: A Step-by-Step Guide | DataCamp,

accessed on September 12, 2025, https://www.datacamp.com/tutorial/llama3-fine-tuning-locally
47. (2025-02-07) Lab Notebook: Load Custom Models in Ollama - MSU HPCC User Documentation, accessed on September 12, 2025, https://docs.icer.msu.edu/2025-02-07-LabNotebook_Load_Custom_models_in_Ollama/
48. How to Customize LLM Models with Ollama's Modelfile - GPU Mart, accessed on September 12, 2025, https://www.gpu-mart.com/blog/custom-llm-models-with-ollama-modelfile
49. Ollama - Building a Custom Model - Unmesh Gundecha, accessed on September 12, 2025, https://unmesh.dev/post/ollama_custom_model/
50. tohur/natsumura-storytelling-rp-1.0-llama-3.1-8b - Featherless.ai, accessed on September 12, 2025, https://featherless.ai/models/tohur/natsumura-storytelling-rp-1.0-llama-3.1-8b
51. Prompt Engineering for AI Guide | Google Cloud, accessed on September 12, 2025, https://cloud.google.com/discover/what-is-prompt-engineering
52. Prompt Engineering Guide, accessed on September 12, 2025, https://www.promptingguide.ai/
53. Prompt Engineering for Game Development - Analytics Vidhya, accessed on September 12, 2025, https://www.analyticsvidhya.com/blog/2024/06/prompt-engineering-for-game-development/
54. Kaggle Game Arena evaluates AI models through games - Google Blog, accessed on September 12, 2025, https://blog.google/technology/ai/kaggle-game-arena/
55. Need help with testing a New AI Game Master (Short Text RPG Session). Seeking Experienced D&D Players : r/RPGcreation - Reddit, accessed on September 12, 2025, https://www.reddit.com/r/RPGcreation/comments/1kh4lu8/need_help_with_testing_a_new_ai_game_master_short/
56. Static Vs. Agentic Game Master AI for Facilitating Solo Role-Playing Experiences - arXiv, accessed on September 12, 2025, https://arxiv.org/html/2502.19519v2
57. Evaluation methods in video game AI competitions | Download Table - ResearchGate, accessed on September 12, 2025, https://www.researchgate.net/figure/Evaluation-methods-in-video-game-AI-competitions_tbl1_322844241
58. Evaluating Story Generation Systems Using Automated Linguistic Analyses - Melissa Roemmele, accessed on September 12, 2025, https://roemmele.github.io/publications/fiction_generation.pdf
59. New Modules for Mothership RPG - Tuesday Knight Games, accessed on September 12, 2025, https://www.tuesdayknightgames.com/collections/mothership-new-modules
60. Featured Modules for Mothership RPG - Tuesday Knight Games, accessed on September 12, 2025, https://www.tuesdayknightgames.com/collections/mothership-featured-modules
61. Adventures & Modules | Mothership - The Largest RPG Download Store! -

DriveThruRPG.com, accessed on September 12, 2025, https://legacy.drivethrurpg.com/browse.php?filters=100003_2110_0_0_0