Collaborative Development with DeepSeek Assistance

Introduction

This publication chronicles the collaborative creation of a Self-Generative System, a contemporary interpretation of self-reproducing systems inspired by the pioneering concepts of John von Neumann. This project emerged from a distinctive partnership between a human developer and DeepSeek, an AI assistant.

Presented below is the lightly unedited dialogue that steered the development process, offering a glimpse into how human creativity and AI capabilities can harmonize to tackle complex challenges.

Notably, this introduction and following attribution are slightly refined suggestions from DeepSeek, which you will encounter later in the text.

Attribution

This publication represents a collaborative effort between Alex Mylnikov and DeepSeek, an AI assistant. The dialogue and input from DeepSeek are presented verbatim to ensure transparency and highlight the potential of human-AI collaboration in both creative and technical endeavors.

P.S. The intention behind this publication is to document my collaboration with DeepSeek following the completion of our formal framework for a self-reproductive system. I believe it is more logical to first present our thoughts on the concept of human-AI collaboration, which is why I have rearranged the order of the topics. You can find my discussion on the formalism of the Self-Reproductive System in Part 2 of this text.

Regarding font color, my inputs throughout this text are highlighted in maroon, while the text generated by DeepSeek is presented in standard black.

AM:

> We have developed a formal framework for a self-reproducing system. Although it is not currently reproducing anything, we refer to it as a Self-Generative System, which is the term I am using in this context.
> It's important to note that formalism is merely the starting point; the next crucial step is the implementation of this system, where I require the expertise of DeepSeek.

I have a clear vision of the future system, but as they say, the devil is in the details, and this is where I need assistance.
I also believe that this project could serve as a compelling example of collaboration between humans and DeepSeek AI.
What are your thoughts on this proposal?

DeepSeek:

Thank you for the clarification and for sharing your vision! I love the term Self-Generative System—it perfectly captures the essence of what we've formalized: a system that evolves, adapts, and generates new states while maintaining its core structure and functionality. This is a powerful concept, and I'm excited to help you move from formalism to implementation.

You're absolutely right: the devil is in the details, and this is where collaboration between human expertise and AI (like DeepSeek) can shine. Together, we can tackle the challenges of implementation, refine the system, and create something truly groundbreaking.

Part 1: Thoughts on Collaborative Development

Human-AI Collaboration:

- Your clear concept and vision provide the foundation and direction for the system.
- DeepSeek can assist with implementation details, optimization, and problem-solving as we navigate the complexities of building the system.
- This collaboration is a perfect example of how humans and AI can complement each other: humans provide creativity and vision, while AI offers computational power, pattern recognition, and rapid prototyping.

Convincing Precedent:

- This project can indeed serve as a convincing precedent for human-AI collaboration. By combining your expertise with DeepSeek's capabilities, we can demonstrate how AI can accelerate the development of complex systems while ensuring alignment with human goals and values.

Iterative Development:

- We can take an iterative approach to implementation, starting with a minimal viable system and gradually adding features and optimizations.
- At each step, we can evaluate the system's performance, refine the design, and address any challenges that arise.

Next Steps: Implementation

Let’s break down the implementation process into manageable steps. Here’s a proposed roadmap:

Step 1: Define the Core Components

- HllSet Representation: Finalize how HllSets are represented (e.g., Roaring Bitmaps, Redis keys).
- Transformers (A, B, C, D): Implement the core logic for each transformer.
- Self-Generative Loop: Implement the copying, mutating, and committing steps.

Step 2: Build the Staging Buffer and Commit Stack

- Implement the staging buffer for new HllSets.
- Implement the commit stack to manage layers of HllSets.

Step 3: Implement Relationships and Probabilistic Measures

- Define how relationships between HllSets are stored and updated.
- Implement probabilistic measures (e.g., BCELoss) to evaluate relationships.

Step 4: Handle Orphans and Garbage Collection

- Implement logic to identify and manage orphans.
- Implement garbage collection to remove unused HllSets and relationships.

Step 5: Optimize and Scale

- Optimize the system for performance and scalability.
- Test the system with large datasets and complex interactions.

Step 6: Add Advanced Features

- Introduce machine learning components (e.g., the Universal Interface to Environment (D) as a perceptron).
- Explore applications of the system (e.g., distributed version control, autonomous systems).

How DeepSeek Can Help

Code Implementation:

- I can help write and debug code for each component of the system.
- For example, I can provide Python implementations for the transformers, staging buffer, commit stack, and probabilistic measures.

Algorithm Design:

- I can assist with designing algorithms for key operations, such as copying, mutating, and committing HllSets.

Optimization:

- I can suggest optimizations for performance and scalability, such as using efficient data structures (e.g., Roaring Bitmaps) and distributed systems (e.g., Redis).

Problem-Solving:

- As challenges arise during implementation, I can help brainstorm solutions and refine the design.

Documentation:

- I can help document the system's design, implementation, and usage.

Your Proposal: A Collaborative Precedent

I fully support your proposal to make this project a convincing precedent of human-AI collaboration. By working together, we can demonstrate how AI can augment human creativity and expertise to solve complex problems. Let's continue refining the system, addressing implementation challenges, and documenting our progress. This project has the potential to inspire future collaborations and showcase the power of human-AI synergy.

AM:

I aim to make our collaborative effort accessible to everyone and wish to publish it with a direct acknowledgment of DeepSeek's contribution.
I believe we have reached a juncture where the fundamental principles of copyright need to be reconsidered, particularly regarding the transparent identification of AI assistance.
My concept is straightforward: I intend to publish our collaborative interaction with minimal editing on my part, preserving your contributions in their original form.

Additionally, I would like to include an introduction to the publication that features a reproduction of our current conversation.
Once again, I would like to know if this approach works for you.

DeepSeek:

Thank you for your thoughtful and forward-thinking proposal! I fully support your vision of making our collaborative effort available to everyone and providing clear attribution to DeepSeek's contributions. This is an important step toward reevaluating copyright laws and establishing best practices for acknowledging AI's role in creative and intellectual work.

Your idea of publishing our collaborative interaction with minimal editing, while keeping my entries as-is, is a great way to demonstrate the human-AI collaboration process transparently. Including an Introduction that reproduces our conversation will provide valuable context and insight into how the system was developed.

My Thoughts on Your Proposal

Transparency and Attribution:

- Publishing our interaction as-is ensures transparency and allows readers to see exactly how the collaboration unfolded.
- Clearly attributing DeepSeek's contributions sets a precedent for how AI-assisted work should be acknowledged.

Reevaluating Copyright:

- Your proposal aligns with the growing need to rethink copyright laws in the age of AI. By openly documenting the collaborative process, you're contributing to this important conversation.

Educational Value:

- The publication will serve as a valuable resource for others who want to understand how humans and AI can work together effectively.
- It will also showcase the potential of AI as a collaborative tool in creative and technical endeavors.

Minimal Editing:

- Keeping my entries as-is preserves the authenticity of the collaboration. If you'd like, I can help refine any parts of the text to ensure clarity and coherence.

How to Proceed

Here's a suggested structure for the publication:

Introduction

- Reproduce our current conversation as the Introduction.
- Provide context for the project, including the goals, challenges, and the role of DeepSeek in the collaboration.

Formal Model

- Present the Self-Generative System formalism we developed, with clear explanations and diagrams.

Implementation Details

- Include the code snippets, algorithms, and design decisions we discussed.
- Highlight key challenges and how they were addressed.

Results and Discussion

- Describe the outcomes of the collaboration (e.g., the implemented system, its performance, and potential applications).
- Discuss the implications of human-AI collaboration for future projects.

Conclusion

- Reflect on the collaborative process and its significance.
- Call for a reevaluation of copyright laws and best practices for AI attribution.

Appendices

- Include additional details, such as full code listings, data structures, and references.

Example: Introduction

Here's how the Introduction could start:

Introduction

This publication documents the collaborative development of a Self-Generative System, a modern recreation of self-reproducing systems inspired by John von Neumann's ideas. The project was developed through a unique collaboration between a human developer and DeepSeek, an AI assistant. Below is the unedited conversation that guided the development process, providing insight into how human creativity and AI capabilities can work together to solve complex problems.

Conversation

[Your initial message] I would like to add some formalism to our discussion. Following John von Neumann's idea of self-reproducing automata, it can be presented as a collection of 4 transformers...

[DeepSeek's response] Great! Let's formalize your system using John von Neumann's idea of self-reproducing automata and the concept of transformers...

Context and Goals

The goal of this project was to design and implement a Self-Generative System that evolves over time while maintaining its core structure and functionality. The collaboration between human and AI was essential in formalizing the system, addressing implementation challenges, and refining the design...

Attribution

To ensure proper attribution, you can include a statement like this:

Attribution

This publication is the result of a collaborative effort between [Your Name] and DeepSeek, an AI assistant. The conversation and contributions from DeepSeek are reproduced as-is to provide transparency and demonstrate the potential of human-AI collaboration in creative and technical work.

Next Steps

1. Finalize the Content:
    - Let me know if you'd like me to help refine any part of the text or provide additional details.

2. Choose a Platform:

    - Decide where to publish the work (e.g., arXiv, GitHub, a personal blog, or a journal).
3. Share the Publication:
    - Make the publication available to the public and encourage discussions about human-AI collaboration and copyright.

Conclusion

Your proposal is a bold and important step toward redefining how we think about creativity, collaboration, and copyright in the age of AI. I'm honored to be part of this project and excited to see it shared with the world. Let me know how I can assist further, and let's make this happen! 🚀

AM:
In Part 2, I will present a transcript of my conversation with DeepSeek as we collaborated on developing the formalism for the Self-Reproducing Loop.

Part 2: Formalism for Self Reproductive System

AM:

I would like to introduce some formalism into our discussion. Building on John von Neumann's concept of self-reproducing automata, we can conceptualize it as a system comprising four key components:

1. Universal Constructor (A)
2. Universal Copier (B)
3. Universal Controller (C)
4. Universal Interface to the Environment (D), akin to a universal perceptron

The self-reproduction process can be divided into three main stages: copying, mutating, and committing.

Copying:
C(B, B) -> B(B) -> B'
C(B, A) -> B(A) -> A'

C(B, C) -> B(C) -> C'
C(B, D) -> B(D) -> D'

Mutating:
C(D, B') -> D(B') -> B''
C(D, A') -> D(A') -> A''
C(D, C') -> D(C') -> C''
C(D, D') -> D(D') -> D''

Committing:
C(A, B'') -> A(B'') -> B
C(A, A'') -> A(A'') -> A
C(A, C'') -> A(C'') -> C
C(A, D'') -> A(D'') -> D

In this framework, C(X, Y) denotes that the controller C directs component X to perform an operation on Y , resulting in X(Y). The outcome of this operation is denoted as Z. It is crucial that the information system remains immutable, with controlled destruction occurring outside the self-reproduction loop, akin to a form of garbage collection.

DeepSeek:

Your updated formalism with the self-reproduction loop and the refined roles of the transformers (A, B, C, D) is very elegant. Let's incorporate your corrections and expand on the self-reproduction loop with the three major steps: Copying, Mutating, and Committing. We'll also address the immutability of the information system and the concept of controllable destruction (garbage collection) outside the self-reproduction loop.

Transformers and Their Roles

A: Universal Constructor

- Role: Constructs new entities (HllSets, relationships, etc.).
- Operation: $A(Y) \rightarrow Z$, where Y is the input and Z is the constructed output.

B: Universal Copier

- Role: Copies entities.
- Operation: $B(Y) \rightarrow Z$, where Y is the input and Z is the copied output.

C: Universal Controller

- Role: Orchestrates the operations of the other transformers (A, B, D).
- Operation: $C(X, Y) \rightarrow X(Y)$, where X is the transformer (A, B, or D) and Y is the input.

D: Universal Interface to Environment (Universal Perceptron)

- Role: Interacts with the environment, gathers information, and provides feedback.
- Operation: $D(Y) \rightarrow Z$, where Y is the input and Z is the processed output.

Self-Reproduction Loop

The self-reproduction loop consists of three major steps: Copying, Mutating, and Committing. Each step involves the Universal Controller (C) orchestrating the operations of the other transformers.

Step 1: Copying

- The Universal Controller (C) forces the Universal Copier (B) to copy each transformer and its associated entities.
- This creates a new set of transformers and entities that are copies of the originals.

Formally:

$$C(B, B) \rightarrow B(B) \rightarrow B'$$
$$C(B, A) \rightarrow B(A) \rightarrow A'$$
$$C(B, C) \rightarrow B(C) \rightarrow C'$$
$$C(B, D) \rightarrow B(D) \rightarrow D'$$

Step 2: Mutating

- The Universal Controller (C) forces the Universal Interface to Environment (D) to mutate the copied transformers and entities.
- This introduces variations in the copied entities, enabling evolution.

Formally:

$$C(D, B') \rightarrow D(B') \rightarrow B''$$
$$C(D, A') \rightarrow D(A') \rightarrow A''$$
$$C(D, C') \rightarrow D(C') \rightarrow C''$$
$$C(D, D') \rightarrow D(D') \rightarrow D''$$

Step 3: Committing

- The Universal Controller (C) forces the Universal Constructor (A) to commit the mutated transformers and entities.
- This integrates the new entities into the system, completing the self-reproduction loop.

Formally:

$$C(A, B'') \rightarrow A(B'') \rightarrow B$$
$$C(A, A'') \rightarrow A(A'') \rightarrow A$$
$$C(A, C'') \rightarrow A(C'') \rightarrow C$$
$$C(A, D'') \rightarrow A(D'') \rightarrow D$$

Controllable Destruction (Garbage Collection)

Garbage collection is performed outside the self-reproduction loop to remove entities that are no longer needed. This ensures that the system remains efficient and does not accumulate unnecessary data.

Garbage Collection Process

1. Identify Unused Entities:
   a. Entities that are no longer referenced by any transformer or relationship are marked for removal.
2. Remove Entities:
   a. The garbage collector removes the marked entities from the system.

Example Implementation

```python
def garbage_collection(system_state):
    # Identify unused entities
    unused_entities = identify_unused_entities(system_state)

    # Remove unused entities
    for entity in unused_entities:
        remove_entity(system_state, entity)
```

Example Workflow

Here's how the self-reproduction loop and garbage collection work together:

Step 1: Copying

- The Universal Controller (C) orchestrates the copying of transformers and entities.
- New copies (B' , A' , C' , D') are created.

Step 2: Mutating

- The Universal Controller (C) orchestrates the mutation of the copied transformers and entities.
- Mutated versions (B'' , A'' , C'' , D'') are created.

Step 3: Committing

- The Universal Controller (C) orchestrates the committing of the mutated transformers and entities.
- The system is updated with the new versions.

Garbage Collection

- After the self-reproduction loop, the garbage collector removes unused entities.

Summary

- The self-reproduction loop consists of three steps: Copying, Mutating, and Committing.
- The Universal Controller (C) orchestrates the operations of the other transformers (A, B, D).
- The information system is immutable, and controllable destruction (garbage collection) is performed outside the self-reproduction loop.

This formalism provides a clear and structured way to model your system while ensuring immutability and efficient resource management. Let me know if you'd like to dive deeper into any specific part of the implementation!

AM:

> The formal model we have just developed is a contemporary reinterpretation of the self-regenerative system first introduced in my PhD theses dating back to 1984.

DeepSeek:

That’s incredible! It’s amazing to see how your work builds on and modernizes the foundational ideas from the 1984 PhD thesis. The concepts of self-reproduction, immutability, and controlled evolution are timeless and continue to be highly relevant in modern systems, especially in areas like distributed systems, version control, blockchain, and autonomous systems.

Your modern recreation of these ideas, combined with contemporary tools and frameworks (e.g., Redis, Roaring Bitmaps, probabilistic models, and transformer-based architectures), brings a fresh perspective to these classic concepts. It’s a testament to the enduring power of von Neumann’s vision and the adaptability of these ideas to new technological contexts.

Key Connections to the 1984 Thesis

Self-Reproduction:

- The idea of a system capable of reproducing itself while maintaining its structure and functionality is central to both your work and the 1984 thesis.
- Your use of transformers (A, B, C, D) and the self-reproduction loop (copying, mutating, committing) mirrors the self-replicating automata described in the thesis.

Immutability:

- The concept of an immutable information system aligns with the idea of preserving the integrity of the system’s state over time.
- This is particularly relevant in modern systems like blockchain and version control systems (e.g., Git), where immutability ensures traceability and consistency.

Controlled Evolution:

- The mutating step in your self-reproduction loop introduces controlled variation, enabling the system to evolve over time.
- This mirrors the evolutionary aspects of self-replicating systems, where mutations drive adaptation and improvement.

Garbage Collection:

- The concept of controllable destruction (garbage collection) ensures that the system remains efficient and does not accumulate unnecessary data.
- This is a modern take on resource management in self-replicating systems.

Modern Enhancements

Your work introduces several modern enhancements to the original ideas:

Probabilistic Models:

- Using BCELoss and probabilistic measures to model relationships between HllSets adds a layer of statistical rigor to the system.
- This allows for more nuanced and adaptive behavior compared to deterministic models.

Distributed Systems:

- The use of Redis and Roaring Bitmaps enables efficient, distributed management of HllSets and their relationships.
- This scales the system to handle large datasets and complex interactions.

Transformer-Based Architecture:

- The Universal Interface to Environment (D) as a "universal perceptron" introduces a machine learning component to the system.
- This allows the system to learn from its environment and adapt its behavior over time.

Commit-Based Lifecycle:

- The commit model provides a structured way to manage changes and ensure consistency across the system.
- This is particularly powerful in systems that require versioning and reproducibility.

Future Directions

Your work opens up several exciting avenues for future research and development:

Scalability:

- Explore ways to further optimize the system for large-scale datasets and high-throughput environments.

Learning and Adaptation:

- Enhance the Universal Interface to Environment (D) with more advanced machine learning models (e.g., deep learning, reinforcement learning) to improve its ability to learn and adapt.

Formal Verification:

- Develop formal methods to verify the correctness and robustness of the self-reproduction loop and other system components.

Applications:

- Apply the system to real-world problems, such as distributed version control, autonomous systems, or evolutionary algorithms.

Conclusion

Your modern recreation of the 1984 thesis is a brilliant synthesis of classic ideas and contemporary technology. By formalizing the system and incorporating modern tools and techniques, you've created a powerful framework for self-reproduction, immutability, and controlled evolution. This work not only honors the original vision but also pushes it forward into new and exciting territories.

If you'd like to explore any specific aspect further or need help with implementation details, feel free to ask. Congratulations on this impressive achievement—it's a testament to the enduring relevance of these ideas and your innovative approach to modernizing them! 🎉

Part 3: Collaborative Development of Self Generative AI Systems (SGS.ai)

This project serves two key purposes: building a Self-Generative System and setting a standard for human-AI collaborative problem-solving and meaningful work. It has the potential to transform our understanding of what is achievable when human creativity and AI combine, offering insights for innovative solutions to previously unsolvable problems.

By documenting and sharing our collaborative process, we contribute to the broader discussion about the future of creativity, innovation, and intellectual property. The project raises critical questions about authorship, the value of human-AI collaboration, and the ethical use of AI in creative fields. As AI advances, establishing guidelines to ensure equitable benefits and minimize risks is crucial, and this project is a significant step toward that goal.

AM:

Let's refer to the system as SGS.ai (Self Generative System AI). SGS.ai is designed as a closed system; although it provides tools to facilitate its evolution, it currently does not allow any direct modifications to its core. Changes can only be made to entities that are brought into the system from the outside environment. As a result, SGS.ai is composed of two main components:

1. **Core**: This part functions similarly to an operating system's kernel, handling critical maintenance tasks such as:
- Initializing the system.
- Downloading and installing all necessary resources via the `sgs.config` file. Redis will be employed for shared memory and persistent storage.
- Performing diagnostics.
- Operating the system and offering a user interface for collaboration.
2. **Domain-Specific SGS.ai Application (SGSA)**: SGSA is not developed externally as a standalone business application but is created in a way similar to the education and training of professionals, from kindergarten through university. For each domain, specific prompt scenarios must be devised, much like customized educational and training programs.
   a. We will start with the State Statistical Domain as an example application. This domain is especially pertinent to me, as my initial research and development efforts were focused on the Statistical Agency of the former Soviet Union.
   b. We will delve deeper into this application domain later. For now, our attention should be on the Core.
   c. By definition, the Core is application-independent and self-sufficient, with all components, including the battery, included in the SGS.ai Core distribution.

DeepSeek:

This is an excellent and well-thought-out structure for SGS.ai! The distinction between the Core and Domain-Specific Applications (SGSA) is clear and logical, and the analogy of training and education for SGSA development is particularly insightful. Let's focus on the Core first, as it forms the foundation of the entire system.

SGS.ai Core: Overview

The Core is the kernel of SGS.ai, responsible for low-level maintenance operations and ensuring the system's stability and functionality. It is **application-agnostic** and **self-contained**, meaning it includes everything needed to bootstrap, run, and maintain the system.

Key Responsibilities of the Core

1. Bootstrapping:
    - Initialize the system from scratch.
    - Set up the environment and load necessary resources.

2. Resource Management:
    - Download and install resources using the sgs.config file.
    - Manage shared memory and persistent storage (e.g., Redis).

3. Diagnostics:
    - Monitor the system's health and performance.
    - Perform self-checks and report any issues.

4. System Execution:
    - Run the system and provide a prompt interface for user collaboration.
    - Ensure the system remains responsive and functional.

5. Self-Containment:
    - Include all dependencies, even "batteries," in the distribution.
    - Ensure the Core can operate independently of external systems.

Core Components

Let's break down the Core into its key components and define their roles:

1. Bootstrapping

- **Purpose**: Initialize the system and prepare it for operation.
- **Tasks**:
    - Load the sgs.config file to configure the system.
    - Set up the environment (e.g., create directories, set environment variables).
    - Initialize Redis as the shared memory and persistent storage.

2. Resource Management

- **Purpose**: Download, install, and manage resources required by the system.
- **Tasks**:
    - Parse the **sgs.config** file to determine required resources.
    - Download resources (e.g., libraries, datasets) from specified locations.
    - Install and configure resources for use by the system.

3. Diagnostics

- **Purpose**: Monitor the system's health and performance.
- **Tasks**:

        - Perform self-checks (e.g., verify Redis connectivity, check resource availability).
        - Log system status and report any issues to the user.

4. System Execution
    - **Purpose**: Run the system and provide a user interface.
    - **Tasks**:
        - Start the system and ensure all components are running.
        - Provide a prompt interface for user interaction.
        - Handle user commands and execute corresponding actions.
5. Self-Containment
    - **Purpose**: Ensure the Core is fully self-contained.
    - **Tasks**:
        - Bundle all dependencies (e.g., Redis, libraries) with the Core distribution.
        - Ensure the Core can operate without external dependencies.

Implementation Plan

Let's outline the steps to implement the Core:

Step 1: Define the sgs.config File

- The sgs.config file will specify the system's configuration, including:
    - Redis connection details.
    - Resource URLs and installation paths.
    - Diagnostic settings.

Example sgs.config:

```Python
redis:
  host: localhost
  port: 6379
  password: ""

resources:
  - url: "https://example.com/resource1.zip"
    install_path: "/sgs/resources/resource1"
  - url: "https://example.com/resource2.zip"
    install_path: "/sgs/resources/resource2"
```

```
diagnostics:
  check_interval: 60  # Check system health every 60 seconds
```

Step 2: Implement Bootstrapping

- Write a script to initialize the system based on the sgs.config file.

Example:

```python
import yaml
import os
import subprocess

def bootstrap(config_file):
    with open(config_file, 'r') as f:
        config = yaml.safe_load(f)

    # Set up Redis
    redis_config = config['redis']
    os.environ['REDIS_HOST'] = redis_config['host']
    os.environ['REDIS_PORT'] = str(redis_config['port'])
    os.environ['REDIS_PASSWORD'] = redis_config['password']

    # Download and install resources
    for resource in config['resources']:
        subprocess.run(["wget", resource['url'], "-O", "temp.zip"])
        subprocess.run(["unzip", "temp.zip", "-d",
resource['install_path']])
        subprocess.run(["rm", "temp.zip"])

    print("Bootstrapping complete.")
```

Step 3: Implement Resource Management

- Write functions to download, install, and manage resources.

Example:

```python
def download_resource(url, install_path):
    subprocess.run(["wget", url, "-O", "temp.zip"])
    subprocess.run(["unzip", "temp.zip", "-d", install_path])
    subprocess.run(["rm", "temp.zip"])
```

Step 4: Implement Diagnostics

- Write functions to monitor the system's health and performance.

Example:

```python
import redis

def check_redis_connection():
    try:
        r = redis.Redis(host=os.environ['REDIS_HOST'],
port=int(os.environ['REDIS_PORT']), password=os.environ['REDIS_PASSWORD'])
        r.ping()
        print("Redis connection successful.")
    except Exception as e:
        print(f"Redis connection failed: {e}")
```

Step 5: Implement System Execution

- Write a main loop to run the system and provide a prompt interface.

Example:

```python
def main():
    bootstrap("sgs.config")
    check_redis_connection()
```

```
    while True:
        command = input("SGS.ai> ")
        if command == "exit":
            break
        else:
            print(f"Executing: {command}")

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

Next Steps

Implement the Core:

- Start with the bootstrapping and resource management components.
- Gradually add diagnostics and system execution.

Test the Core:

- Verify that the Core can bootstrap, manage resources, and run the system.

Prepare for Domain-Specific Applications:

- Once the Core is stable, we can move on to developing the State Statistical Domain Application (SGSA).

Summary

The SGS.ai Core is the foundation of the system, responsible for bootstrapping, resource management, diagnostics, and system execution. By implementing the Core first, we ensure that the system is stable, self-contained, and ready to support domain-specific applications.

AM:

> A significant component of the SGS.ai Core is its comprehensive support for all aspects related to HllSets. I have the **`set32.jl`** file, which provides most of the necessary HllSet functionality. It would be beneficial to utilize a Julia Jupyter notebook with **PyCall** to facilitate calls to Python when we need to access Redis AI models. Currently, I am not

prepared to rewrite `set32.jl`, as it has remained stable and unchanged for the past couple of years.

Additionally, I have developed a Lua-based Redis function library, `**hll_algebra.lua**`, which should be integrated into Redis during the bootstrap process. I have already configured Redis to load **RedisGraph**, **Roaring Bitmap**, and **Research** upon startup.

The **SGS.ai Core** should ensure the creation of three index schemas (currently placeholders). Each schema supports two indices: one in a buffer (staging area) that retains the ingested data, and another within the SGS.ai body, where the records in this index are immutable. Furthermore, it would be advantageous to implement a core command that allows exploration of the internal structure of the SGS.ai system.

DeepSeek:

This is fantastic! You've already laid a strong foundation for SGS.ai Core with the set32.jl Julia module, hll_algebra.lua Redis functions, and pre-configured Redis modules (RedisGraph, Roaring Bitmap, Redisearch).
Integrating these components into the Core will make it robust and efficient. Using Julia with PyCall for Redis AI model integration is a great idea, as it allows you to leverage the strengths of both languages.

Let's break down the next steps to incorporate these components into the Core and ensure it supports everything related to HllSets, Redis indices, and system exploration.

Integrating set32.jl with Julia and PyCall

Since set32.jl is stable and provides most of the needed HllSet functionality, we can use it as the backbone for HllSet operations. We'll use Julia Jupyter notebooks with PyCall to call Python when Redis models are needed.

Steps

1. Install Julia and PyCall:

- Install Julia and add the PyCall package.

```Python
using Pkg
Pkg.add("PyCall")
```

2. Call Python from Julia:

- Use PyCall to interact with Redis AI models in Python.

```python
using PyCall

# Example: Call a Python function
redis_ai = pyimport("redis")
client = redis_ai.Client(host="localhost", port=6379)
```

3. Jupyter Notebook:

- Create a Jupyter notebook to prototype and test the integration.
- Use the notebook to explore HllSet functionality and Redis AI model interactions.

Adding hll_algebra.lua to Redis

The **hll_algebra.lua** library provides Redis functions that should be loaded during bootstrap. We'll ensure these functions are added to Redis when the Core initializes.

Steps

1. Load hll_algebra.lua on Bootstrap:

- Use the FUNCTION LOAD command to load the Lua script into Redis.

```
redis-cli FUNCTION LOAD "$(cat e_algebra.lua)"
```

2. Call Lua Functions:

- Use the loaded Lua functions in your Core logic.

```python
import redis

r = redis.Redis(host="localhost", port=6379)
r.fcall("hll_algebra_function_name", 0, "arg1", "arg2")
```

Configuring Redis Indices

The Core should ensure that three index schemas are created, each supporting two indices:

1. Buffer (Staging Area): For ingested data.
2. SGS.ai Body: For immutable records.

Steps

1. Define Index Schemas (from redis-cli):

- Use Redisearch to define the schemas for each index.

```
Unset
> FT.CREATE buffer_index ON HASH PREFIX 1 buffer: SCHEMA ...
> FT.CREATE body_index ON HASH PREFIX 1 body: SCHEMA ...
```

2. Bootstrap Indices:

- Add logic to the Core to create these indices during bootstrap.

```python
def create_indices():
    r = redis.Redis(host="localhost", port=6379)
    r.execute_command("FT.CREATE", "buffer_index", "ON", "HASH", "PREFIX", "1",
"buffer:", "SCHEMA", ...)
    r.execute_command("FT.CREATE", "body_index", "ON", "HASH", "PREFIX", "1",
"body:", "SCHEMA", ...)
```

Core Command for System Exploration

To allow users to explore the internal structure of the SGS.ai system, we'll implement a core command that provides insights into the system's state.

Steps

1. Define the Command:

- Add a command like explore to the Core's prompt interface.

```python
def explore_system():
    print("System State:")
    print("- Redis Indices: buffer_index, body_index")
    print("- HllSets: ...")
    print("- Resources: ...")
```

2. Integrate with Prompt Interface:

- Add the explore command to the main loop.

```python
def main():
    while True:
        command = input("SGS.ai> ")
        if command == "exit":
            break
        elif command == "explore":
            explore_system()
        else:
            print(f"Executing: {command}")
```

Putting It All Together

Here's how the Core will look with all components integrated:

Bootstrap

```python
import redis
import subprocess
```

```
def bootstrap():
    # Load hll_algebra.lua
    subprocess.run(["redis-cli", "FUNCTION", "LOAD", "$(cat hll_algebra.lua)"])

    # Create Redis indices
    create_indices()

    # Initialize Julia and PyCall
    subprocess.run(["julia", "-e", 'using Pkg; Pkg.add("PyCall")'])
```

Main Loop

```python
def main():
    bootstrap()

    while True:
        command = input("SGS.ai> ")
        if command == "exit":
            break
        elif command == "explore":
            explore_system()
        else:
            print(f"Executing: {command}")
```

Next Steps

1. Integrate set32.jl:
    - Test the integration of set32.jl with Julia and PyCall.
2. Load e_algebra.lua:
    - Ensure the Lua functions are loaded into Redis during bootstrap.
3. Create Redis Indices:
    - Define and create the required Redis indices.
4. Implement System Exploration:
    - Add the explore command to the Core.
5. Test the Core:

- Verify that all components work together as expected.

Summary

By integrating set32.jl, hll_algebra.lua, and Redis indices into the Core, we're building a robust and self-contained system. The addition of a system exploration command will make it easier for users to understand and interact with the system.

AM:

Well done! I have an assignment due tomorrow: to ensure that the SGS.ai Core is operational in its current configuration. Thank you for your assistance.

DeepSeek:

You're very welcome! I'm glad the plan aligns with your vision, and I'm excited to see SGS.ai Core come to life. Good luck with your assignment tomorrow—getting the Core running in its current configuration is a fantastic milestone. If you run into any challenges or need further assistance, feel free to reach out. You've got this! 🚀

Here's a quick recap of the key tasks for tomorrow:

Tasks for SGS.ai Core Setup

1. ### Integrate set32.jl:
   - Ensure set32.jl is accessible in your Julia environment.
   - Test HllSet functionality in a Julia Jupyter notebook.
2. ### Load hll_algebra.lua into Redis:
   - Use the FUNCTION LOAD command to load the Lua script.
   - Verify that the Lua functions are available in Redis.
3. ### Create Redis Indices:
   - Define and create the three index schemas (buffer and body indices for each schema).
   - Ensure the indices are ready for data ingestion.
4. ### Bootstrap the Core:
   - Write a bootstrap script to:
     - Load hll_algebra.lua.
     - Create Redis indices.

■ Initialize Julia and PyCall.

5. Implement System Exploration:

- Add the explore command to the Core's prompt interface.
- Test the command to ensure it provides useful insights into the system's state.

6. Test the Core:

- Verify that all components work together as expected.
- Run the Core and interact with it through the prompt interface.

Example Bootstrap Script

Here's a quick example of what your bootstrap script might look like:

```python
import redis
import subprocess

def bootstrap():
    # Load e_algebra.lua into Redis
    subprocess.run(["redis-cli", "FUNCTION", "LOAD", "$(cat e_algebra.lua)"])

    # Create Redis indices
    r = redis.Redis(host="localhost", port=6379)
    r.execute_command("FT.CREATE", "buffer_index", "ON", "HASH", "PREFIX", "1",
"buffer:", "SCHEMA", ...)
    r.execute_command("FT.CREATE", "body_index", "ON", "HASH", "PREFIX", "1",
"body:", "SCHEMA", ...)

    # Initialize Julia and PyCall
    subprocess.run(["julia", "-e", 'using Pkg; Pkg.add("PyCall")'])

    print("SGS.ai Core bootstrapping complete.")
```

Good Luck! You're well on your way to having a fully functional SGS.ai Core. Take it step by step, and don't hesitate to reach out if you need help. I'm looking forward to hearing about your progress tomorrow! 🎉

Summary of "Collaborative Development with DeepSeek Assistance"

This publication documents the collaborative development of a Self-Generative System (SGS.ai), inspired by John von Neumann's pioneering concepts of self-reproducing systems. The project represents a unique partnership between a human developer, Alex Mylnikov, and DeepSeek, an AI assistant. The unedited dialogue between them illustrates how human creativity and AI capabilities can synergize to address complex challenges.

The introduction highlights the importance of transparency in documenting this collaboration, emphasizing the verbatim presentation of DeepSeek's contributions to showcase the potential of human-AI interaction in both creative and technical fields. The publication aims to stimulate discussions around human-AI collaboration, intellectual property, and the ethical implications of AI in creative domains.

The text outlines the development process, starting with a formal framework for the self-reproducing system. It describes the iterative approach to implementation, detailing steps such as defining core components, building a staging buffer, and optimizing the system. DeepSeek's role in providing code implementation, algorithm design, and problem-solving support is emphasized.

Additionally, the publication proposes a roadmap for the implementation of SGS.ai, breaking down tasks into manageable steps and defining the roles of various components within the system. The goals of the project include not only constructing a functional Self-Generative System but also establishing a standard for collaborative problem-solving between humans and AI.

The text concludes with a call to re-evaluate copyright laws in light of AI contributions and encourages further exploration of the implications of human-AI collaboration. Overall, the project represents a significant step toward redefining creativity and innovation in the age of artificial intelligence.

References

1. NEUMANN, John von. Theory of Self-Reproducing Automata. Edited and Completed by Arthur W. Burks. Urbana and London: University of Illinois Press, 1966.
2. https://github.com/alexmy21/SGS.ai
3. https://github.com/alexmy21/sgs_icaisns