Author: Alex Mylnikov, PhD in computer science and statistics

email: alexmy@lisa-park.com

Affiliation: Lisa Park Inc

Key words:

AI Models

Technology

Innovation

Self Generative Systems

JohnVonNeumann

Automation

FutureTech

Self Generative Systems (SGS) and Its Integration with AI Models

This article begins by exploring the concept of self-reproducing automata, as introduced by John von Neumann (refer to [1]). According to the Wikipedia entry [15], the initial studies on self-reproducing automata date back to 1940. Over the span of more than 80 years, significant advancements have been made in this field of research and development, bringing us closer to the realization of systems capable of self-reproduction, which we will refer to as self-generative systems (SGS).

0. Introduction to John von Neumann theory of self-reproduction

John von Neumann's concept of self-replicating systems is intriguingly straightforward. Imagine a system composed of three distinct modules: A, B, and C.

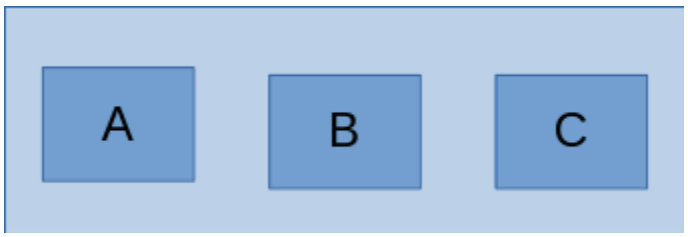

Figure 1. Base John von Neumann Self Reproducing System

Module A acts as a Universal Constructor, capable of crafting any entity based on a provided a blueprint or schema. Module B functions as a Universal Copier, able to replicate any entity's detailed blueprint or duplicate an entity instance. Module C, the Universal Controller, initiates an endless cycle of self-replication by activating modules A and B.

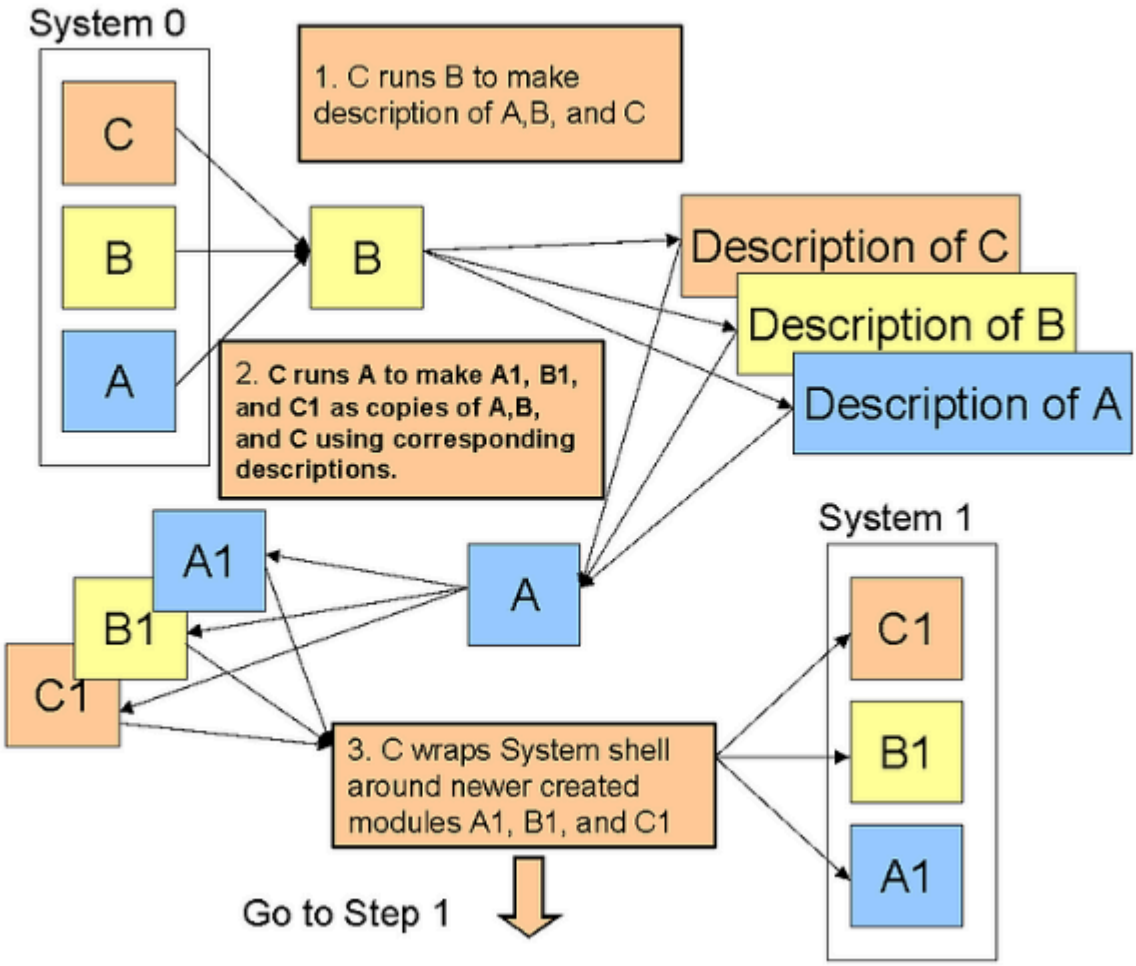

Figure 2. Flow-chart of Self Reproduction

The self-replication process begins with Module C, leading to the creation of an identical system, System 1, from the original System 0. This new system is equally capable of initiating its own self-replication cycle, adhering to the same algorithm.

From this analysis, several key insights emerge.

Firstly, the self-replication algorithm is sufficiently generic to be implemented across various platforms.
Secondly, Module C's infinite loop can orchestrate the self-replication process.
Lastly, this algorithm represents a theoretical framework for system upgrade automation, or self-upgrading. However, in its basic form, a self-replicating system

merely clones itself. To enhance its utility, a fourth module, Module D, is introduced. This module enables interaction with the system's environment and access to its resources, effectively functioning as an application within an operating system composed of Modules A, B, and C.

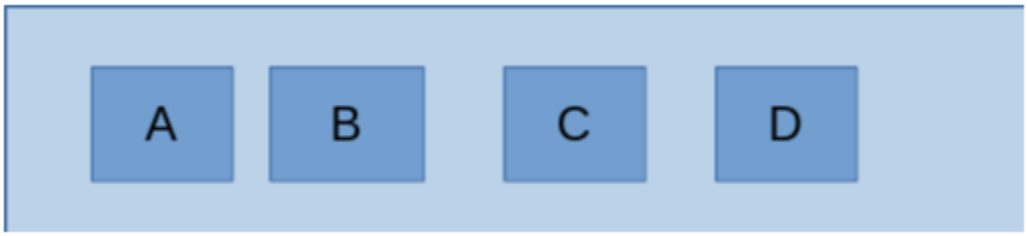

Figure 3. Extended Self Reproducing System

Additionally, a special unit for storing module descriptions, termed the System Description is incorporated. This upgraded self-replication process, depicted in subsequent figures, involves creating copies of each module description (A, B, C, D) alongside the System Description unit. This leads to the creation of an upgraded system version, which then replaces the old version, thus achieving a new iteration of the system.

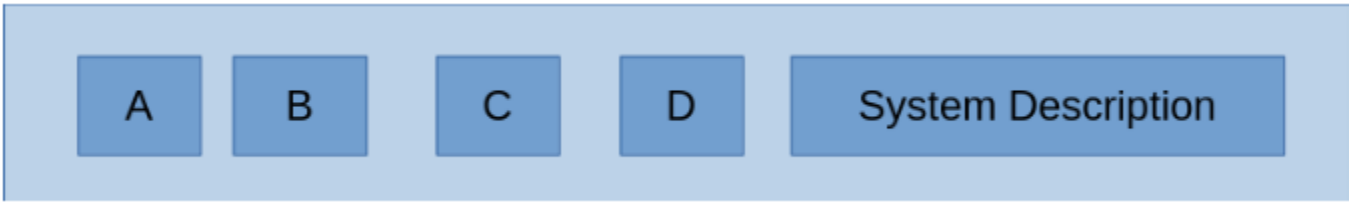

Figure 4. Self Reproducing System with System Description

This enhanced model differs from John von Neumann's original concept by introducing a dedicated unit for system descriptions, allowing the system to interact with its environment via Module D, and modifying the role of Module B to work solely with the System's Description. Despite these advancements, the initial creation of the first self-replicating system remains an unsolved "Chicken and Egg" dilemma. Yet, as we draw parallels between this abstract model and software systems, we see opportunities for applying self-replication in managing software application life cycles.

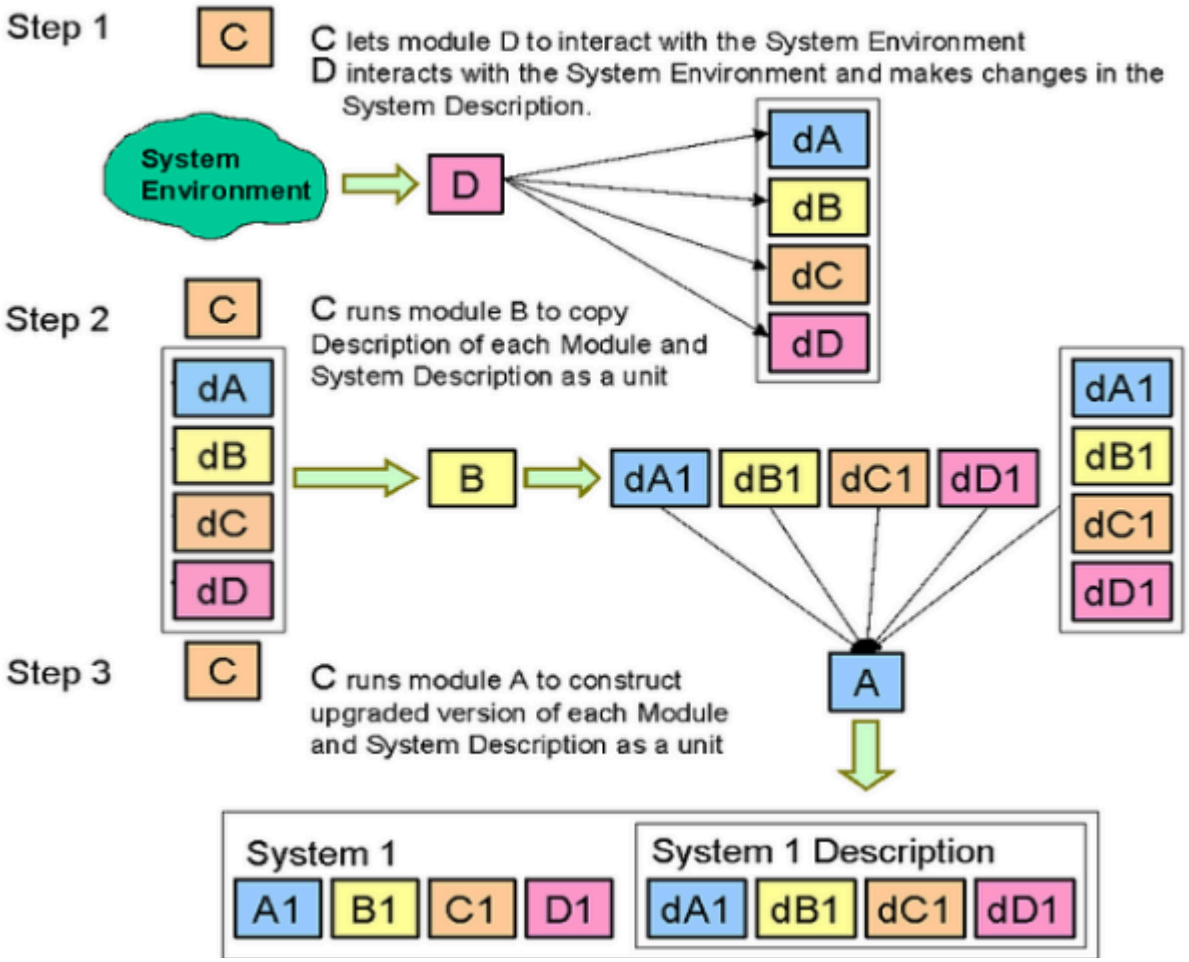

Figure 5. Self Reproduction flow-chart for extended Self Reproducing System with System description

In software terms, Modules A, B, and C could represent engines facilitating continuous service processes, such as database engines, servers, or runtime environments. Module A could serve as a compiler or interpreter, generating processes based on source code. Module B might support reflection, serialization, and data buffering, ensuring system persistence and enabling development, evolution, and backup. Module D would represent application software, facilitating user and environment interaction. Ultimately, viewing self-generative systems (SGS) as a means to standardize and automate the development cycle of software applications—from development to testing, and testing to production—opens up exciting possibilities for autonomous software development. The purpose of this document is to utilize the concept of SGS within the Metadata Management System to analyze the Socio-Economic System.

In summary, Self-Generative Systems (SGS) integrate the principles of John von Neumann's Self-Reproducing Automata into artificial intelligence models. This research and development initiative distinguishes itself from traditional methodologies in several key ways:

1. **Adaptation of the HyperLogLog Algorithm**: We utilize the HyperLogLog method to approximate datasets within fixed-size structures, which mimic the properties and operations of conventional sets, such as union, intersection, and complement. These structures are known as HllSets.

2. **Emphasis on Metadata Over Raw Data**: By employing HllSets, we represent metadata and link it to the corresponding data approximated as HllSets. This approach transitions the focus from direct data manipulation to a strategy centered on metadata.

3. **Implementation of a Self-Generative Loop**: Drawing inspiration from John von Neumann's theory, we have developed a self-generative loop within the AI model. This innovative framework is a core component of our project, Self-Generative Systems (SGS). Further details can be found in Chapter 3: Life Cycle, Transactions, and Commits.

1. Introduction to Metadata

Metadata is essentially information about information. It encompasses any type of digital content that can be stored on a computer, including documents, databases, images, videos, audio files, and sensor signals. From a metadata standpoint, all of these forms of data are treated equally and hold the same significance.

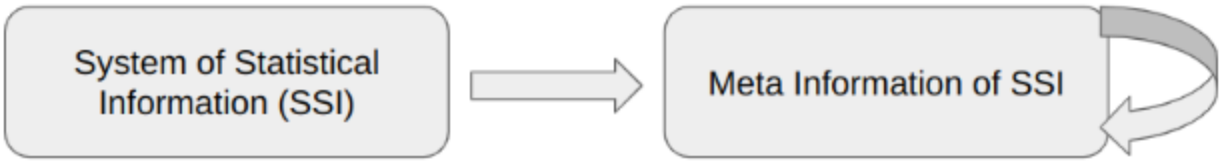

Figure 6. Mapping Statistical Information to Meta Information

What, then, can be said about the concept of metadata for metadata? Essentially, metadata refers to data about data. Thus, when we discuss metadata derived from metadata, we are essentially discussing the same entity.
This point is crucial. We propose to manage both the metadata of original data and the metadata of metadata through a singular metadata system. This approach is visually represented in the figure, where we depict the metadata loop closing in on itself.
Furthermore, we delineate the ongoing process of generating metadata, which evolves over time both from the initial data and from metadata previously created. This cyclical process highlights the dynamic and iterative nature of metadata

generation.

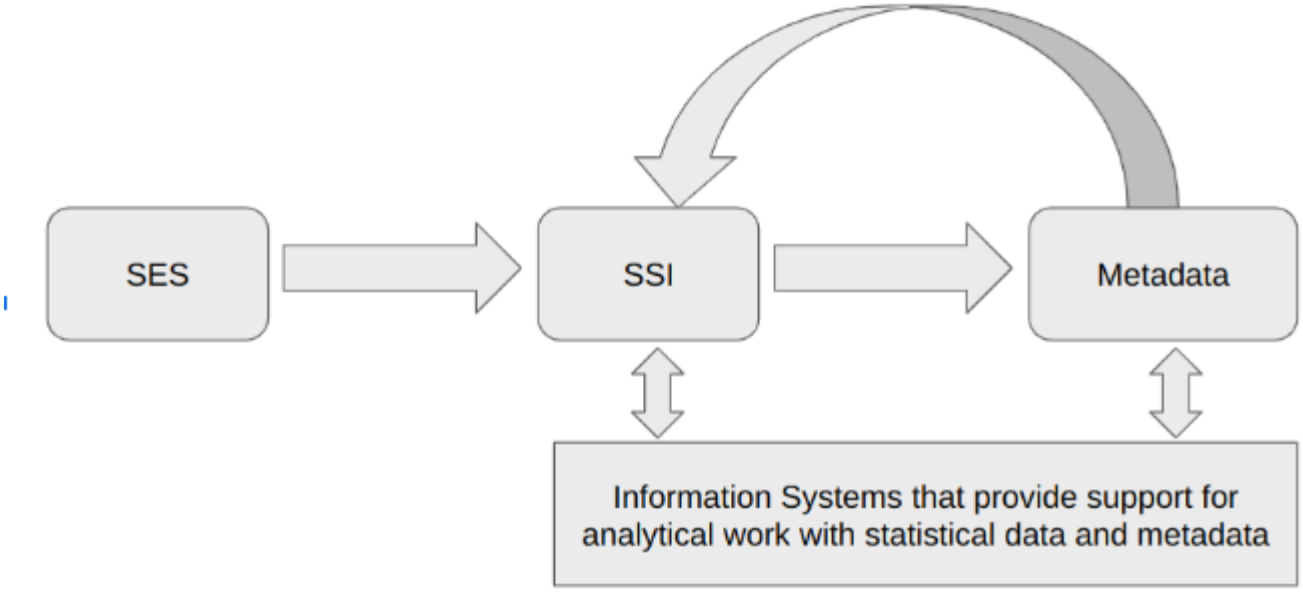

Figure 7. Feed back loop of the SSI into Matadata

By integrating socio-economic systems (SES) mapping into statistical information systems (SIS) through statistical observation, and then mapping SIS into Metadata, we can develop a comprehensive and generalized scheme.

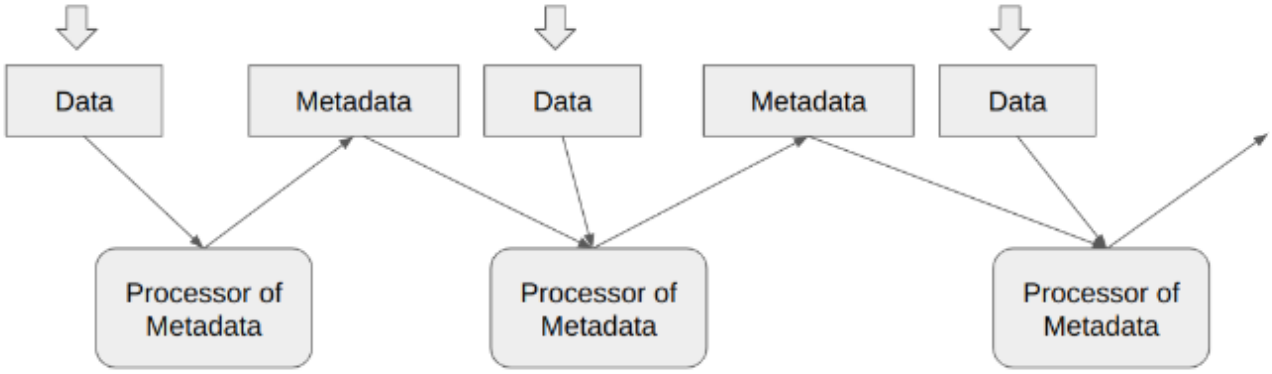

Figure 8. Flow-chart of sequential Data processing and mapping Data to Metadata

The diagram illustrates the SES SIS and SIS (Metadata), clearly demonstrating how it maintains the integrity and structural relationships within the displayed system, as discussed earlier.
When it comes to metadata, it encompasses all forms of data. For these datasets, it's imperative to create descriptive metadata elements and to forge connections among these elements.
Conceptually, metadata can be visualized as a graph. Within this graph, metadata

elements are depicted as nodes, while the links between these elements are represented by the graph's edges. This structure facilitates a comprehensive and interconnected representation of metadata, enhancing the understanding and utilization of statistical information.

2. HllSets

HllSets is a data structure based on the HyperLogLog algorithm developed by Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier [6]. We significantly refined this approach by developing a data structure that, while maintaining the compactness of the original HyperLogLog structure, supports all the standard properties of Set Theory. In the post [3], we demonstrated that HllSets adhere to all the fundamental properties of Set theory.
The fundamental properties that HllSets complies with are as follows: Commutative Property:
1. $(A \cup B) = (B \cup A)$
2. $(A \cap B) = (B \cap A)$
Associative Property:
3. $(A \cup B) \cup C) = (A \cup (B \cup C))$
4. $(A \cap B) \cap C) = (A \cap (B \cap C))$
Distributive Property:
5. $((A \cup B) \cap C) = (A \cap C) \cup (B \cap C)$
6. $((A \cap B) \cup C) = (A \cup C) \cap (B \cup C)$
Identity:
7. $(A \cup \varnothing) = A$
8. $(A \cap U) = A$
In addition to these fundamental properties, HllSets also satisfy the following additional laws:
Idempotent Laws:
9. $(A \cup A) = A$
10. $(A \cap U) = A$

To see the source code that proves HllSets satisfies all of these requirements, refer to hll_sets.ipynb[8].

3. Life cycle, Transactions, and Commits

This section will delve into some key technical details that are crucial for developing SGS as a programming system.

3.1. Transactions

In this section, we employ the "transaction" index (or a transactional table - t_table, if we're discussing databases) as an alternative to the System Description found in

the self-reproduction diagram of Chapter 0 (refer to Figure 9). The following is a flowchart that outlines the process of handling external data in the Metadatum SGS.

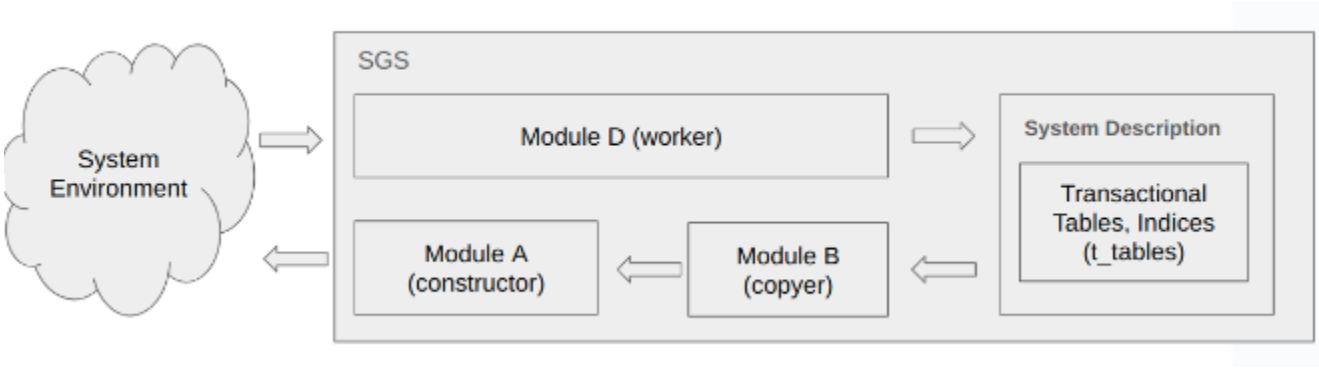

Figure 9. Interaction between SGS and Environment

Module D obtains inputs from the System Environment and records these inputs by generating records in the "transaction" index. Simultaneously, Module A, with assistance from Module B (the copier), retrieves these references from the "transaction" index. It then processes these references by engaging the appropriate processors and subsequently uploads the processed data back into the System.
It is crucial to note that SGS never directly sends incoming data to the System. Instead, it first segregates all incoming data logically into a staging area using the references in the "transaction" index.
This approach helps us achieve several objectives:
1. Clear separation between data already present in the System and new data.
2. Complete control over the processing of new data, enabling us to track completed tasks and pending work. It also facilitates support for parallel processing and recovery from failures.
3. Ability to isolate unsupported data types.
4. Strict adherence to the self-reproduction flow outlined in Chapter 0.

3.2. Commits

In the SGS (Self Generative System), each entity instance is categorized under one of three primary commit statuses, which are crucial for tracking modifications.

These statuses are as follows:
1. Head: This status signifies that the entity instance represents the most recent modification.
2. Tail: An instance with this status is identified as a prior modification, indicating that it is not the latest version.
3. Deleted: This status is assigned to instances that have been marked as removed from the system.
To better understand how commit statuses function, consider the following illustration. The diagrams visualize the timeline of modifications, starting from the

most recent (current time) at the top and progressing downwards to the earliest at the bottom.

Current time.

	Commit ID	Time	item_1	item_2	item_3	item_4	item_5	item_6
1	bcdea	3					head	head
2	cdeab	2		head	head			
3	abcde	1	head	tail		head		

Time of the previous commit. Take note of how the updated version of item_2 changed the commit state in row 2 from its original state.

	Commit ID	Time	item_1	item_2	item_3	item_4		
1	cdeab	2		head	head			
2	abcde	1	head	tail		head		

Time of the initial commit.

	Commit ID	Time	item_1	item_2		item_4		
1	abcde	1	head	head		head		

Essentially, each commit in the system carries its unique "commit forest," depicted through a distinct matrix. For every commit, there's a designated matrix. However, there's no cause for concern—these matrices are virtual. They don't need to exist physically as we can generate them as needed.

At time = 1, we observed three items: item_1, item_2, and item_4, all of which were tagged with the 'head' status, indicating their current and active state.

By time = 2, changes were made to item_2. Consequently, at this juncture, a new version of item_2 emerged within the SGS, introducing a fresh element. This new version was also tagged with the 'head' status, while the previous version's status was switched to 'tail,' indicating it's now a historical entry.

Updating is a methodical process that entails several steps:

- Creating a new version of the item to be updated;
- Applying the required modifications to this new version;
- Saving these changes;
- Establishing a connection between the new and the former version of the item.

By time = 3, two additional elements—item_5 and item_6—were introduced and similarly tagged with the 'head' status.

This mechanism of commits in the SGS crafts a narrative of system evolution. Each cycle of self-reproduction within the SGS adds new chapters to this "history book," linking back to system snapshots at the time of each commit.

In this "history book," we distinguish between 'head' and 'tail.' The 'head' represents the immediate memory and the current state of the SGS, while the 'tail' serves as an archive. Although still accessible, retrieving information from the 'tail' requires additional steps.

The commit history functions as the intrinsic timeline of Self-Generative Systems, akin to biological time in living organisms.

3.3. Static and Dynamic Metadata Structure

Data serves as a mirror of the real world, while metadata, as an abstraction of the data, serves as a reflection of the real world.

Figure 10. Sequential process of converting observations into data and data to the metadata

These observations rest on the underlying belief that there is a direct correspondence between the complexities of the real world and the elements within our datasets. Here, each piece of data is essentially a combination of a value and its associated attributes. When we define associative relationships based on data and its metadata, drawing on similarities between data elements, we're dealing with what's known as a **Static Data Structure**. The term "static" implies that these relationships are fixed; they remain constant and can be replicated as long as the data elements are described using the same attributes. Modern databases excel at mapping out these types of relationships.

Nonetheless, the primary aim of data analysis is to unearth hidden connections among data elements, thereby uncovering analogous relationships in the real world. This endeavor presupposes that the relationships we discover within our data somehow mirror those in the real world. However, these connections are not immediately apparent—they are hidden and transient, emerging under specific conditions. As circumstances evolve, so too do the relationships among real-world

elements, necessitating updates in our Data Structure to accurately reflect these changes. This leads us to the concept of a Dynamic Data Structure.

A **Dynamic Data Structure** emerges from the process of data analysis, facilitated by various analytical models, including Machine Learning or Artificial Intelligence. Broadly speaking, an analytical model comprises an algorithm, source data, and the resulting data. The relationships forged by these models are ephemeral and might not have real-world counterparts. Often, they represent an analyst's subjective

interpretation of the real world's intricacies. These model-generated relationships constitute a Dynamic Data Structure.
The nature of a Dynamic Data Structure is inherently fluid—relationships deemed accurate yesterday may no longer hold today. Different models will vary in their relevance, and the analyst's primary challenge is to select the models that best fit the current real-world scenario and the specific aspects under consideration.

SGS AI Architecture

To fully grasp the concepts and solutions discussed in this section, it is essential to revisit the definitions of the key building blocks of SGS provided in Appendix 2.
The diagram below illustrates the advanced architecture of Self-Generative Systems (SGS) with an integrated Large Language Model (LLM). This representation highlights a seamless and non-intrusive method of integrating AI models, particularly LLMs, into the SGS framework, functioning in a plug-and-play manner. Notably, both the Metadata Models (MM) and the Large Language Models (LLM) receive inputs from a shared tokenization process. This commonality ensures that both components process the same foundational data, facilitating efficient and coherent system performance. This integration exemplifies how modern AI components can be effectively incorporated into broader systems to enhance functionality and adaptability.

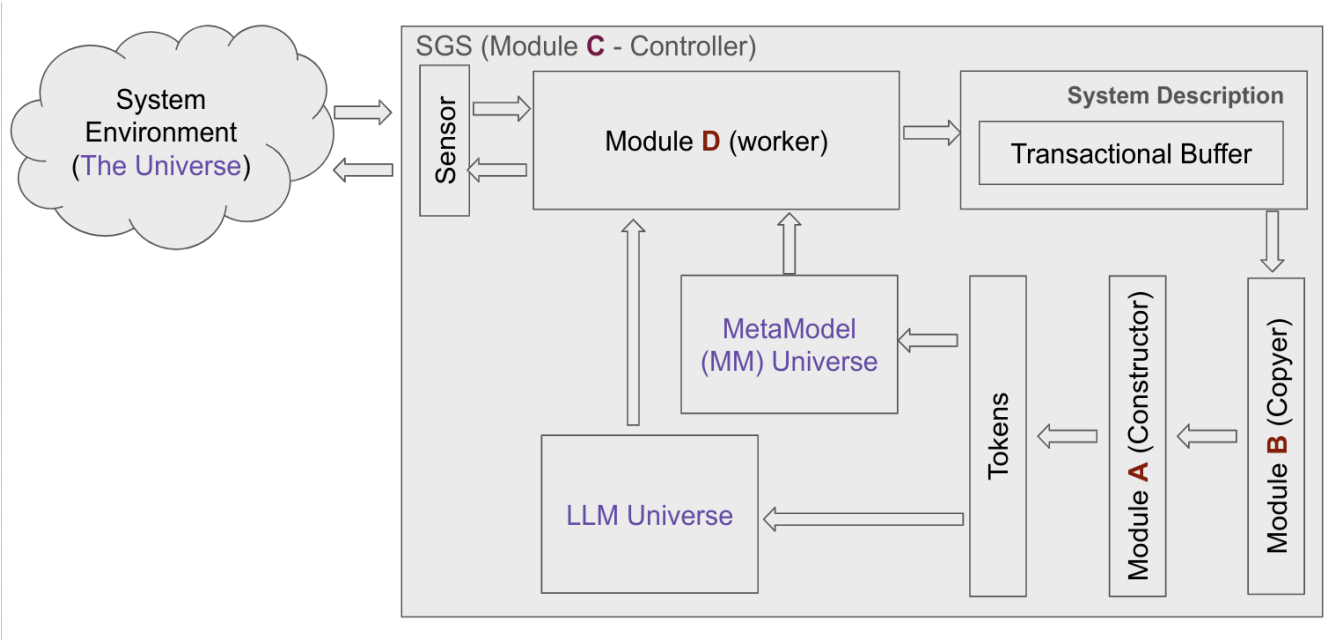

Figure 11. Interactions between SGS and Environment (The Universe)

The next diagram presented below illustrates the natural symbiosis between Metadata Models (MM) and Large Language Models (LLM), showcasing how they complement each other effectively within a system. As observed, MM operates on acquired data and primarily leverages analytical tools and techniques, including the application of high-level set operations (HllSet). These models are inherently more grounded, focusing on realistic, pragmatic outcomes derived from concrete data insights.
In contrast, LLMs, like other AI models, depend on the synthesis of new ideas by tapping into their deep understanding of the relationships between elements within their domain. These models are characterized by their creativity and idealism, often producing innovative yet sometimes unrealistic outputs or even prone to generating hallucinatory results.

As highlighted in the diagram, MM serves a critical role in balancing the creative exuberance of LLMs. By applying reasonable constraints, MM can harness and refine the imaginative outputs of LLMs, grounding them in practicality. This interplay ensures that the strengths of both models are utilized to their fullest, combining creativity with realism to produce robust, reliable, and useful results. This symbiotic relationship not only enhances the functionality of each model but also significantly improves the overall efficacy of the system in which they are integrated.

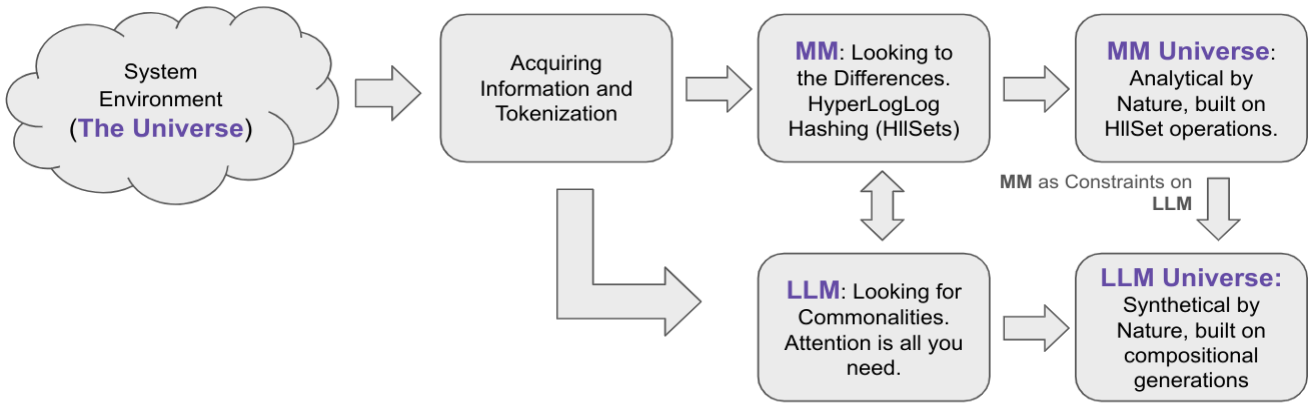

Figure 12. Flow-chart of transforming Environment Observations into data and then into LLM and MM

MM: Looking at the Differences. HyperLogLog Hashing (HllSets). MM Universe: Analytical by Nature, built on HllSet operations.
The MM universe is fundamentally analytical in nature, relying on a structured approach to understanding and manipulating data. Metadata models serve as explicit constraints that guide the generation process. Through HllSet operations,

LLM: Looking for Commonalities. Attention is all you need. LLM Universe: Synthetical by Nature, built on compositional generations.
The LLM universe is synthetical by nature, focusing on the identification of commonalities rather than differences. Grounded in the principles of attention mechanisms, LLMs leverage vast amounts of textual data to generate human-like text through compositional generation. This approach enables LLMs to synthesize information from diverse sources, creating coherent narratives or responses based on patterns learned during training.

While MM emphasizes analytical differentiation, LLM seeks to establish connections and similarities across datasets. This synthesis is driven by the model's ability to attend to various parts of the input data, allowing it to weave together disparate pieces of information into a unified output. However, this compositional generation process is not without its challenges; it requires careful calibration to ensure that the generated content remains relevant and meaningful.

The Role of Metadata Models as Constraints

The integration of metadata models into the generative processes of LLMs can enhance their effectiveness by providing a structured framework that guides the synthesis of information. By imposing explicit constraints, metadata models can help mitigate issues related to coherence and relevance, ensuring that the outputs generated by LLMs adhere to the desired characteristics of the intended application.

For instance, in a self-generative system that seeks to create personalized recommendations, metadata models can define parameters such as user preferences and contextual information. These constraints can guide the LLM in synthesizing outputs that are not only relevant but also tailored to the specific needs of the user.

In summary, the interplay between self-generative systems, metadata models, and large language models highlights the importance of both analytical and synthetical approaches in the generation of meaningful outputs. While MM emphasizes the need for explicit constraints through HllSet operations, LLM focuses on the synthesis of commonalities through attention mechanisms. By integrating these paradigms, we can create robust self-generative systems capable of producing high-quality, contextually relevant content.

Summary of Self Generative Systems (SGS) and Its Integration with AI Models

This article explores the concept of Self Generative Systems (SGS), drawing inspiration from the foundational work of John von Neumann on self-reproducing automata.

John von Neumann's theory of self-reproduction describes a system composed of four modules: A (Universal Constructor), B (Universal Copier), C (Universal Controller), and D (environment-interacting module). This enhanced system allows for self-replication with the ability to interact with its surroundings, paving the way for practical applications in software development and lifecycle management.

A core aspect of this exploration is the concept of metadata, which serves as information about information. The article emphasizes the importance of managing both the metadata of original data and the metadata of metadata through a unified system, illustrating the cyclical nature of metadata generation.

The HllSets data structure, based on the HyperLogLog algorithm, is introduced as a means to satisfy fundamental properties of set theory while facilitating efficient data analysis. Additionally, the article discusses the significance of transactions and commits in the lifecycle of data within the SGS, highlighting how these mechanisms enable effective data management and historical tracking.
The architectural integration of SGS with AI models, particularly Large Language Models (LLMs), are also presented. This symbiosis allows for a seamless and non-intrusive connection between metadata models (MM) and LLMs, leveraging the strengths of both to produce more robust and contextually relevant outputs. The MM universe focuses on analytical differentiation, while the LLM universe emphasizes synthetical generation, making their collaboration crucial for effective

data processing.
The provided code demonstrates the ease of integrating SGS.ai with AI models, emphasizing the synchronization of metadata with model data. This integration brings several advantages:
1. Enhanced Efficiency: Automating data synchronization minimizes manual effort and accelerates the model training process.
2. Improved Accuracy: Continuous updating of data ensures AI models reflect real-world changes, enhancing output reliability.
3. Scalability: SGS.ai can manage increased data loads, making it suitable for applications of any size.
4. Flexibility: The integration supports a wide range of AI models and data types, allowing for diverse analytical applications.
5. Cost-Effectiveness: Streamlined data management reduces operational costs, enabling organizations to focus on innovation.

In conclusion, the synergy between SGS.ai and AI platforms, facilitated by straightforward integration, enhances the overall functionality and adaptability of data analytics environments. The complementary relationship between MM and LLM approaches highlights the potential for creating powerful, self-generative systems capable of producing high-quality, contextually relevant content.

Concepts and advantages presented in the article, showcasing the transformative potential of SGS and its integration with AI technologies.

Appendix 1

PoC: SGS simulation using Enron emails as a source. Day-by-day processing

The source code for the PoC can be found on:
https://github.com/alexmy21/lisa_meta/blob/main/lisa_enron.ipynb

This implementation demonstrated that integrating SGS.ai with various AI models is remarkably straightforward. The primary task of this integration involves synchronizing the data within SGS metadata with the data used in specific AI models.

Appendix 2.

SGS

- Self-Reproductive System (SRS): A theoretical construct inspired by the work of John von Neumann, referring to systems capable of self-reproduction. These systems consist of modules that can create duplicates of themselves, enabling continuous cycles of replication and evolution.
- Self-Generative Systems (SGS): A subclass of self-reproducing systems that are designed to automate and standardize software development cycles. SGS facilitates the continuous development, testing, and production of software applications by leveraging self-replicating algorithms and metadata management.
- Module A (Universal Constructor): In the context of self-reproducing systems, this module is responsible for creating any entity based on a provided blueprint or schema, effectively acting as a constructor within the system. Module B (Universal Copier): This module is tasked with replicating any entity's blueprint or duplicating an instance of an entity. It ensures that the necessary information for replication is available.
- Module C (Universal Controller): A module that initiates and manages the self-replication process within a self-reproducing system, activating Modules A and B to create duplicates of the system.
- Module D (Environmental Interaction Module): An enhancement to the basic self-replicating system, this module enables interaction with the system's environment and manages access to external resources.
- System Description: A dedicated unit within a self-generative system that stores descriptions of each module, allowing for better interaction with the environment and facilitating upgrades and modifications.
- Commit Status: In the context of SGS, this refers to the state of an entity instance in the system, which can be categorized as 'Head' (the most recent modification), 'Tail' (prior modifications), or 'Deleted' (marked for removal).
- Transactional buffer (Index, table): A table or index used within the SGS to manage incoming data and track the processing of new data, ensuring clear separation between existing and new data while facilitating recovery and parallel processing.
- Static Data Structure: A type of data structure that defines fixed relationships among data elements that remain constant over time. These structures are typically used in traditional databases.
- Dynamic Data Structure: A data structure that evolves over time based on analysis and changing conditions, reflecting the transient nature of relationships among real-world elements.
- Commit Function: A function within the SGS that processes updates to the system, categorizing changes and managing the transition of data from the 'head' (current state) to the 'tail' (archived state) as part of the system's evolution.

HllSet and Entity

- HllSet: A data structure based on the HyperLogLog algorithm, used to efficiently approximate the cardinality (number of distinct elements) of large datasets. HllSets adhere to fundamental properties of set theory including commutative, associative, distributive properties, and identity laws.
- Node: In the context of graph databases, a node represents an entity or a data point. In the case of HllSet Relational Algebra, each node represents an HllSet, characterized by attributes such as SHA1 hash identifiers and cardinality, among others.
- Edge: A connection between two nodes in a graph database. Each edge has attributes such as source, target, type of relationship (labeled by r_type), and additional properties in JSON format.
- Token: A basic unit of data within the system dictionary of the graph database. Each token is represented by a unique hash ID, a binary representation, and its frequency within the datasets. It also keeps references to all HllSets that include this token.
- SHA1 hash: A cryptographic hash function used to generate unique identifiers for data points (nodes) in HllSets. It ensures that each node can be uniquely identified based on its content.
- Projection: In the context of HllSet Relational Algebra, projection refers to the operation of mapping one set of HllSets onto another, typically representing the intersection of HllSets corresponding to rows and columns in a tabular structure.
- Union, Intersection, Complement, XOR: Set operations applied to HllSets to form new nodes in the graph database, each resulting in a new HllSet node that is added to the graph.
- Entity: Introduced in entity.jl , this structure encapsulates metadata using HllSets. It represents a key component in the SGS, where metadata is managed and manipulated as entities rather than traditional numerical embeddings.
- Graph: Defined in graph.jl , this structure uses Entity instances as nodes and represents connections between these nodes as edges. It facilitates operations on sets within a graph-based architecture, enhancing the handling and processing of interconnected data.
- Static Structure operations: Operations that do not alter existing instances of an entity but facilitate the creation of new instances through various set operations like union, intersection, and complement.
- Dynamic Structure operations: Operations that support modifications to an entity instance while adhering to the principle of immutability. These operations help manage changes within the nodes of the neural network and the relationships among them, crucial for the self-reproducing capabilities of SGS.
- Advance operation: A key operation in SGS that facilitates the self-reproduction of neural network elements by calculating changes through added, retained, and deleted subsets within HllSets. This operation is essential for predicting and managing the future state of entities within SGS.

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://en.wikipedia.org/wiki/Kozma_Prutkov
3. https://www.linkedin.com/posts/alex-mylnikov-5b037620_hllset-relational-algebra-activity-7199801896079945728-4_bI?utm_source=share&utm_medium=member_desktop
4. https://www.linkedin.com/posts/alex-mylnikov-5b037620_hyperloglog-based-approximation-for-very-activity-7191569868381380608-CocQ?utm_source=share&utm_medium=member_desktop
5. https://www.linkedin.com/posts/alex-mylnikov-5b037620_hllset-analytics-activity-7191854234538061825-z_ep?utm_source=share&utm_medium=member_desktop
6. https://algo.inria.fr/flajolet/Publications/FlFuGaMe07.pdf
7. https://static.googleusercontent.com/media/research.google.com/en//pubs/a
8. https://github.com/alexmy21/SGS/blob/sgs_ai_32/hll_sets.ipynb
9. https://www.linkedin.com/posts/alex-mylnikov-5b037620_demo-application-enron-email-analysis-with-activity-7195832040548614145-5Ot5?utm_source=share&utm_medium=member_desktop
10. https://github.com/alexmy21/lisa_meta/blob/main/lisa_enron.ipynb
11. https://github.com/alexmy21/lisa_meta/blob/main/hll_algebra.ipynb
12. https://arxiv.org/pdf/2311.00537 (Machine Learning Without a Processor: Emergent Learning in a Nonlinear Electronic Metamaterial)
13. https://s3.amazonaws.com/arena-attachments/736945/19af465bc3fcf3c8d5249713cd586b28.pdf (Deep listening)
14. https://www.deeplistening.rpi.edu/deep-listening/
15. https://en.wikipedia.org/wiki/Von_Neumann_universal_constructorhttps://en
16. https://github.com/alexmy21/SGS/blob/sgs_ai_32/simulation.ipynb