**1. CodeDoctorBrainBrane**

* **From dev -> scope, map dependencies, zip -> github -> trigger file upload to openai -> replace file on CodeDoctorBrainBrane**
* **From live -> send message -> CodeDoctorBrainBrane**
* **Execute Coding Chain -> Xpoll BrainBrane**
* **While coding, eval all code in docker sandbox**
* **If it evals in docker sandbox, add it to program sim docker codebase and then simulate testing code in docker program sim (which is aggregating snippet files for codebase)**
* **When complete, self-zip the project with dependency maps and create coding brainbrane for any new parts**
* **Update CodeDoctorBrainBrane zip**

**2. Dynamic Workflows (Application Builder)**

* **Builder for Chains and Pipelines**
* **Creates ChainTypes and PipelineTypes**
  + **Procreate**
  + **ProcreateTypes**
  + **Chat Types**
  + **Chat Modes**
  + **Agent Organs**

**3. Kanban**

**4. Autonomous Tasking**

**5. Zettelkasten**

**6. Dynamic Tool Creation and Compilation**

* **Upload an API ref**
* **Create CodeBrainBrane**
* **Get back a swarm of CodeDoctorBrainBrane and Xpolls with dependency maps and all workflows figured out**
* **(Automatically add frontend buttons)**

**7. Dynamic Frontend**

* **Ability for CodeDoctorBrainBrane to send magic commands thru json tools that go to CodeBrainBranes that create features on the current frontend**
* **Automatically able to code your own app and save it as its own shard**
* **Then revert the view to default**

**From Egregore Templates ->**

**8. Linking JSONs with JSON-LD Ontology Frames**

* **What is JSON-LD?** JSON-LD (JSON for Linked Data) is a way to express linked data using JSON. It provides a mechanism to add semantic context and relationships to your JSON objects, making them machine-understandable.
* **Creating Ontology Frames:** For each of your JSON objects (Egregore Profiles, CAWs, etc.), create corresponding JSON-LD ontology frames. These frames will define:
  + **Classes:** Represent the types of entities in your system (e.g., AiEgregoreProfile, CascadingAccordionWeb).
  + **Properties:** Describe the attributes and relationships of these entities (e.g., hasDomain, hasSubdomains, isPartOf).
  + **Vocabularies:** Use standard vocabularies (e.g., Schema.org, Dublin Core, domain-specific ontologies) as well as your own custom terms.

**9. Creating a Meta-Ontology**

* **Purpose:** A meta-ontology describes the overarching concepts, relationships, and rules governing your entire AI Egregore system. It unifies the individual ontologies you've created.
* **Content:** Your meta-ontology might include classes like "Ontology", "Class", "Property", and relationships like "inheritsFrom", "hasInstance", etc.
* **Tools:** Consider using ontology development tools like Protégé to help you build and visualize your meta-ontology.

**10. Building a Python Interpreter**

* **Purpose:** The interpreter will translate your ontology structures into executable Python code, essentially bringing your Egregore system to life.
* **Challenges:** This can be complex, as you'll need to map ontological concepts to Python classes, functions, and data structures. Consider potential approaches:
  + **Rule-based:** Create rules that translate ontology patterns into Python code snippets.
  + **Library-based:** Develop a Python library with pre-defined classes and functions that mirror your ontology.
  + **Hybrid:** A combination of rules and a library for flexibility.

**11. Encapsulating Implications & Building Programmatically**

* **Function Creation:** Once you have a way to generate basic Python code, start encapsulating more complex logic:
  + **Mecha Creation:** Write functions like build\_mecha() that leverage the ontological definitions to create the necessary components and relationships.
  + **Other Operations:** Add functions to manipulate CAWs, execute ToOTs, manage DUOs, and more.
* **Programmatic Generation:** Expand your capabilities to build new components or entire Egregore profiles programmatically based on defined ontologies.

**12. Flow Builder**

**13. Game GUI Overlay**

* **RTS**

**14. Cryptocurrency for gas**

**15. Skins and themes (allegories/sanctuaries/bizzi beehive/gyms/etc etc)**

**n. Semantic Embedding Analysis (DSPy)**

* **Input/Output Signatures:** Define how to represent the inputs and outputs of your AI system as semantic tokens (e.g., using concept embeddings derived from your ontology).
* **Neo4j Snapshots:** Utilize Neo4j, a graph database, to store and query the relationships between your semantic tokens, creating snapshots of the system's reasoning patterns.
* **Neural Network Training:** Instead of a traditional language model, train a neural network (e.g., Graph Neural Network) to:
  + **Identify Patterns:** Look for recurring patterns or correlations within the semantic embeddings and Neo4j snapshots.
  + **Heatmap Generation:** Visualize areas of high activity or connection density, aiding in understanding the Egregore's reasoning process.

N+1. Self-fine tune