Project Genesis: The Generic SDLC: Applying STL Principles to AI-Native Engineering

Date: February 24, 2026 **Theme**: Architectural Restatement of the AI SDLC (v2.8)

The Crisis of the "Agentic" Pipeline

In the first wave of AI-assisted software engineering, we attempted to model the SDLC as a sequence of specialized personas. We built "Requirements Agents," "Design Agents," and "Coding Agents," each with its own hard-coded rules and prompts. This was the "Object-Oriented" phase of AI SDLC—a collection of monolithic classes, each heavily coupled to a specific stage of the development pipeline.

This model failed for the same reason monolithic OO designs often fail: **rigidity**. If you wanted to add a new asset type (like a Security Audit or an API Spec), you had to build a new agent. If the technology stack shifted, every agent's "expertise" had to be manually retrained.

The breakthrough came when we stopped viewing the SDLC as a sequence of people and started viewing it as **Generic Programming**.

The STL Analogy: Orthogonality in the SDLC

The C++ Standard Template Library (STL), pioneered by Alexander Stepanov, revolutionized software by decoupling **Algorithms** from **Data Structures**. It proved that an algorithm like `std::sort` didn't need to know *what* it was sorting, provided the data structure exposed a standard interface (Iterators) and a comparator (Predicates).

The AI SDLC (v2.8) is the application of this "Generic" principle to the entire engineering lifecycle.

1. The Algorithm: The Universal `iterate()` Engine

In our model, there is only **one operation**: `iterate()`. This is the universal engine of the methodology. It is "blind" to the domain. It doesn't know if it is writing a React component or a high-level design document. It only knows how to sense a **Delta** (δ) between a current state and a target state, and how to produce a new candidate that reduces that delta.

2. The Container: The Asset Graph

The "Data Structures" are our **Asset Types** (Intent, Requirements, Design, Code). These are nodes in a directed cyclic graph. Instead of travelling through a pipeline, a software feature is a **Composite Vector**—a trajectory through this graph. The graph is zoomable and extensible; you can "fold" or "unfold" its complexity without changing the underlying algorithm.

3. The Iterator: Admissible Transitions

Iterators in the STL provide a way to traverse containers. In the AI SDLC, **Edges** define the admissible transitions between assets. They bridge the gap between "Design" and "Code," providing the algorithm with the specific context needed to move from one node to the next.

4. The Universal Functor: The Properly Constrained LLM

This is the heart of the system. In the STL, you pass a "Functor" or "Predicate" to an algorithm to define the logic of comparison. In the AI SDLC, the **LLM is the Universal Functor**.

Because the LLM can reason over non-numeric data, it acts as a **Generic Predicate** that can evaluate any edge. However, an unconstrained LLM is prone to hallucination (probability degeneracy). To make it a "Properly Constrained Evaluator," we surround it with a **Constraint Surface** (the `Context[]`): * **REQ Keys**: Provide the coordinate system for traceability. * **ADRs**: Define the formal boundary conditions. * **Markov Criteria**: Define the objective "stop condition" for stability.

By parameterizing the universal `iterate()` engine with these LLM-driven functors, we achieve a system that can process any "type" of software artifact with formal rigor.

Functor Escalation: $F_D o F_P o F_H$

One of the most powerful features of generic programming is the ability to specialize algorithms for performance. Our design implements this through **Natural Transformations** of the functor:

- **Deterministic** (F_D): When ambiguity is zero, we use cheap, fast "specialized" functors like compilers and linters.
- **Probabilistic** (F_P): When ambiguity exists, we escalate to the "generic" LLM functor.
- **Human** (F_H): When ambiguity is persistent, we escalate to the human judgment functor.

This "Escalation Chain" ensures the system is as fast as a traditional build tool when things are clear, and as thoughtful as an architect when things are ambiguous.

Projections: The Methodology as a Generator

Finally, the generic model allows for **Projections**. Just as a C++ template is only instantiated when needed, the AI SDLC is a **Methodology Generator**.

By choosing a **Profile** (e.g., “Minimal” or “Standard”), you are essentially “compiling” a specific version of the methodology. A “Minimal” projection might collapse the Requirements and Design nodes into a single edge, using only the Agent functor. A “Full” projection expands the graph to include 24/7 Sensory Monitoring and Multi-Agent Coordination.

The logic is **Logically Complete** in the blueprint, but only as complex as the project demands at runtime.

Conclusion: From Tools to Organisms

The shift from specialized agents to **Generic SDLC Programming** marks the transition from “AI Tools” to “Homeostatic Systems.” By decoupling the **Process** (Iteration) from the **Constraint Surface** (Spec + Context), we have built a methodology that doesn’t just assist the developer—it regulates itself.

In this new paradigm, the engineering task is no longer “writing code.” It is **shaping the constraint surface** so that the Universal Evaluator can converge the Asset Graph toward a stable, high-quality product.