Siloquy – Introduction

---

1 Motivation

1.1 The End of Human-Written Code

Something fundamental is changing in software engineering. For decades, the limiting factor in building software systems was the pace at which skilled human developers could write and review code. That limit no longer holds.

AI coding systems are already capable of generating well-structured, correct, production-quality code across all major programming languages – not over days or weeks, but within seconds. The transition is not gradual: AI-generated code is not a supplement to human coding. It is becoming the primary mode of software production.

This shift has an immediate and dramatic consequence: **the volume of code in the world will explode**. Systems that took teams of hundreds of engineers several years to build can now be produced in a fraction of that time, at a fraction of that cost. Code is no longer a scarce resource. The infrastructure and algorithms that already run the digital world – plus everything being built on top of it – will be implemented, extended, and transformed at a velocity no prior engineering organization could have achieved.

1.2 The New Role of Human Owners

When AI systems write the code, the nature of human work changes entirely.

The engineers, architects, and domain experts who previously spent most of their time writing and reviewing implementation are now in a fundamentally different position: they define the vision, the needs, and the meaning of the system. They decide what the software should accomplish and why. They set the direction, resolve ambiguities, and judge the outcome.

But this creates an immediate problem. **Humans and AI systems need to communicate at a common level.** The human owner speaks in terms of purpose and intent. The AI system works in terms of code. For this exchange to be productive – not just for today's task, but for a system's entire lifetime – there must be a shared, stable layer that both sides can use to reason about what the software means, what it does, and what it is for.

That layer cannot be code. Code is what the AI produces, not what humans meaningfully reason about. The shared layer must sit above code: at the level of **architecture, meaning, and intention**.

1.3 The Weight of Existing Systems

The software that already runs the world did not wait for this transition. It exists now, in millions of lines of code, accumulated over decades. The algorithms driving financial systems, healthcare infrastructure, industrial control, and global communication are buried inside highly

complex, evolving codebases that no single human fully understands. The teams th
at originally
built them have dispersed. The documentation is outdated. The implicit knowledge
 has been
forgotten.

This is the deeper challenge: **existing software cannot simply be replaced**. B
efore any AI
system can meaningfully evolve, refactor, or extend a large legacy system, it mu
st understand
what that system already does — not at a surface level, but in precise, structur
ed, navigable
detail. That understanding cannot be obtained by reading outdated documents. It
 cannot be
approximated by asking an AI to analyze individual files one at a time. It requi
res a systematic,
persistent, machine-readable representation of meaning: a model that says not ju
st what lines of
code are present, but what each part of the system *does* and why it *exists*.

1.4 Understanding as a Prerequisite for Progress

With AI, a long-needed wave of software improvement finally becomes possible. Sy
stems that have accumulated years of technical debt can be restructured. Functio
nality that was never fully
documented can be made explicit. Behavior that exists only in the minds of retir
ed engineers can
be recovered, formalized, and made navigable.

But none of this can happen without a thorough, shared understanding of meaning
 and intention.
An AI system that does not know *why* a component exists will not know when to p
reserve it,
when to restructure it, or when to replace it. A human owner who cannot see the
 meaning of the
code they direct cannot judge whether the AI is taking the system in the right d
irection.

**Understanding is not a prerequisite only for humans. It is a prerequisite for
 any intelligent
system operating on software.** The stakes are high: the systems being evolved a
re not toys.
They are the infrastructure of modern life.

---

2 Approach

2.1 The Meta-Language Already Exists

Programming languages are formal languages. They define a precise grammar of str
uctures —
functions, conditions, loops, calls, modules, classes, interfaces — that every s
oftware engineer
understands, regardless of the specific language they work in most. This grammar
 is not
arbitrary. It has been refined over decades of practice, standardization, and sh
ared experience.
It is the meta-language of software: a set of concepts that every specialist in
 the field holds
in common.

Any higher-level representation of software must take this seriously. It cannot afford to be
purely theoretical, operating in abstract constructs that have no grounding in what is actually
implemented. If the concepts used to describe software at a higher level are disconnected from
the concepts used to implement it, the representation becomes a fiction — useful neither for
human reasoning nor for AI processing.

The right approach is the opposite: **build the higher-level model as a structured superset of
what can be implemented**. Every concept at the abstract level must map to something real in the code below it. Every relationship must be traceable. The higher level adds structure and meaning; it does not replace implementation reality with something imaginary.

2.2 Why Diagrams Were Never Enough — And Why That Changes Now

Graphical modeling languages like UML have existed for decades. They are well-defined,
well-understood, and backed by a large body of accumulated software engineering knowledge. They provide exactly the kind of language-neutral, structure-preserving abstractions that software systems need.

Yet they have never become the primary way that large software systems are built. The reason is
straightforward: **creating diagrams was more difficult and more constraining than writing code**. Keeping diagrams synchronized with implementation required effort that produced no immediate value. Developers who could express a solution in five lines of code saw no benefit in first drawing a ten-box UML diagram to describe what they already knew how to write. The tools were cumbersome. The process was slow. The result was that UML diagrams were produced for
documentation purposes, usually after the fact, and then abandoned as the system evolved.

This constraint is now removed. With AI, a human can describe their intention in any form that
comes naturally — in natural language, in rough sketches, in partial specifications — and refine
it interactively. The AI can translate that intention into formal structure, generate the
implementation, and maintain the alignment between the two. The human is no longer responsible for the mechanical labor of keeping diagrams and code synchronized. They are responsible only for the judgment: does this match what I meant?

2.3 Siloquy's Approach

UML, extended where necessary, provides the proven structural vocabulary for the higher-level
model. It is understood by both humans and AI systems. It maps cleanly onto the constructs of
real programming languages. It is the right foundation for a shared representation.

Siloquy's approach is to build this foundation systematically:

1. **Extract the full structural reality of existing software** into a formal, language-neutral
   graph representation — grounded in the meta-language of programming, not in any single
   language's syntax.

2. **Reduce that representation into a stable, ordered semantic structure** – one that can be
   evaluated, explained, and navigated deterministically, independent of implementation detail.

3. **Annotate the structure with meaning and intention** – human- and AI-readable descriptions of what each part does (meaning, derived bottom-up from implementation) and why each part exists (intention, derived top-down from system purpose).

4. **Expose all of this through a unified access model** – one that serves both human engineers
   exploring the system visually and AI agents traversing it programmatically, using the same
   underlying representation.

The result is a **persistent, navigable, machine-readable semantic model of software** – one
that enables both understanding and creation of software-based systems with unrivaled speed and precision.

---

3 What is Siloquy

Siloquy is an initiative to create a **semantic digital twin of software systems** – existing
and new – by transforming source code into a **structured, navigable, and machine-interpretable representation** derived directly from the implementation.

At its core, Siloquy produces a **layered model** connecting four levels of abstraction:

- **Structural Layer** – the concrete implementation: source files, functions, control flow,
  call relationships, variables, conditions, and organizational containment. Captured
  faithfully and exhaustively, in a language-neutral form.

- **Abstraction Layer** – a reduced, dependency-ordered graph of **MeaningCells**: uniform
  semantic units produced by collapsing the structural layer. The abstraction layer eliminates
  incidental complexity while preserving all essential relationships. It establishes the order
  in which the system can be understood.

- **Semantic Layer** – two orthogonal annotations carried by every MeaningCell:
  - **Meaning** – what this part of the system does, derived bottom-up from implementation.
  - **Intention** – why this part exists, derived top-down from system purpose and boundary
    truth.

- **Boundary Layer** – the system's interface with the external world: libraries, databases,
  sensors, external services. Boundary truth anchors the top-down annotation of intention
  and makes the system's external dependencies explicit.

These layers are **explicitly linked**. Every element at a higher level is groun

ded in elements
at the level below it, making bidirectional navigation possible: from high-level intention down
to individual code lines, and from any code fragment back up to its role in the overall system.

Siloquy is designed as a **backend and backbone**, not as a monolithic application. Its
representation can be consumed by human-facing visual frontends, by AI agents traversing the
model programmatically, and by analytical tools operating above raw source code. Because all
layers share a single graph, context is preserved across abstraction boundaries – something
neither plain source code nor traditional documentation can provide.

---

4 Design Outline

Siloquy is organized into five sequential Layers, each producing a well-defined structure that
the next Layer consumes. The Layers are designed to be composable, deterministic, and
language-independent.

4.1 Layer 1 – Extraction of Structure

Layer 1 ingests source code and constructs a **complete, faithful, language-neutral structural
graph** of the system. No semantic interpretation is performed at this stage. The goal is
structural completeness: every element of the implementation must be represented, and every
relationship must be traceable to its source.

To manage complexity without mixing concerns, Layer 1 is organized along **five orthogonal
Axes**, each capturing a distinct structural dimension:

Axis	Structural Dimension
A – Lexical / Structural Flow	How control flows inside a lexical region: sequences, branches, loops
B – Unit Connectivity	How functions, methods, and units call and depend on each other
C – Data and Condition Structure	How variables and conditions shape state and predicate dependencies
D – Asynchronous Execution	How execution can occur outside normal call/return; synchronization
E – Structural Containment	Where code lives: modules, files, classes, repositories; inheritance

Each Axis is internally coherent and can reference but does not merge with the others. The
combined output of Layer 1 is a multi-dimensional structural graph – the **Software Code
Intermediate Representation (SCIR)** – stored persistently in a graph database. This is the
foundation on which all subsequent Layers operate.

4.2 Layer 2 — Abstraction of Structure

Layer 2 reduces the detailed structural graph from Layer 1 into a **single, ordered semantic structure** suitable for systematic evaluation. The central question Layer 2 answers is:

> *In what order can the parts of this system be meaningfully explained?*

Layer 2 collapses all structural artifacts into a uniform node type: the **MeaningCell**. From Layer 2 onward, there is only one kind of node in the graph. All structure is encoded as relationships between MeaningCells.

Layer 2 produces **two complementary graph structures** over the same set of MeaningCells:

- **MDG — Meaning Dependency Graph**: a directed acyclic graph expressing which MeaningCells must be understood before others. Edges carry a `role` attribute that captures the nature of the dependency (sequential contribution, conditional branch, variable reads/writes, boundary calls). The MDG defines the bottom-up evaluation order used by Layer 3 and the top-down propagation order used by Layer 4.

- **TOM — Threads of Meaning**: the logical execution paths through the system, expressed over MeaningCells. TOM captures the temporal and causal flow of execution — entry points, branches, calls, returns — without the incidental complexity of the structural graph.

Both structures are derived from the same Layer 1 artifacts and together constitute the complete precondition for semantic annotation.

4.3 Layer 3 — Extraction of Meaning

Layer 3 is a **pure annotation pass**. It adds no new nodes or edges. It does not modify the structure produced by Layer 2. Its sole function is to populate a `meaning` field on every MeaningCell.

`meaning` answers: *What does this part of the system do?*

Meaning is derived **bottom-up**: finer-grained MeaningCells are evaluated before the coarser-grained cells that depend on them, following the evaluation order established by the MDG. Each cell's meaning is synthesized from the meanings of its dependencies plus the structural evidence available from Layer 1.

Different MeaningCell kinds are annotated according to the kind of structure they represent:

- **Block MeaningCells** — the primary execution regions; their meaning is the core semantic unit at each level.

- **Variable MeaningCells** – contribute explanatory context when read by a Block; drive
  meaning when written.
- **Condition MeaningCells** – express the predicate that governs a decision or loop.
- **Decision and Loop MeaningCells** – integrate the meanings of their branches and bodies.
- **Structural MeaningCells** – propagate composed meaning upward from Units to Files, Classes,
  Modules, and the Repository.

Layer 3 also recognizes **External Units** (libraries and frameworks used by the system) and
Implied External Entities (databases, services, sensors, and other boundary targets that
exist beyond the system's code horizon). Both are represented as MeaningCells and annotated
accordingly. The `touches` edge role records the relationship between a boundary call and the
implied entity it communicates with.

4.4 Layer 4 – Extraction of Intention

Layer 4 performs the **reverse pass**: it traverses the MDG top-down and populates an
`intention` field on every MeaningCell.

`intention` answers: *Why does this part of the system exist?*

Where Layer 3 derived meaning from implementation upward, Layer 4 propagates purpose downward:
from the highest-level intention anchors, through intermediate structural levels, down to
individual execution blocks and variables.

Layer 4 introduces a set of **stereotypes** that classify MeaningCells by their role in the
intentional structure:

Stereotype	Role
`<<TOP_INTENTION>>`	Top-level anchors: the root purposes of the system
`<<EXTERNAL_UNIT>>`	External libraries already recognized in Layer 2/3
`<<BOUNDARY_CALL>>`	API and I/O calls that cross the system boundary
`<<IMPLIED_ENTITY>>`	Virtual entities beyond the code horizon (databases, services, sensors)

The only structural addition Layer 4 makes is the creation of `<<IMPLIED_ENTITY>>` MeaningCells at the system boundary, connected to their corresponding boundary calls by `CONTRIBUTES_TO` edges. All other structures from Layers 1–3 are read-only.

The result is a fully dual-annotated graph: every MeaningCell carries both a `meaning` (what it
does) and an `intention` (why it exists). These two fields are the primary semantic payload of
the Siloquy knowledge base.
4.5 Layer V1 – Siloquy Access Model (SAM)

Layer V1 defines the **Siloquy Access Model (SAM)**: the formal specification of

 how humans and
AI agents access, navigate, and interpret the knowledge produced by Layers 1–4.

SAM is not a user interface. It is a **projection and navigation model** that de
fines:
- how subgraphs are selected from the full knowledge base,
- how selected subgraphs are projected into view-type representations,
- how users and agents navigate between levels of abstraction.

SAM exposes the Siloquy knowledge base as **four navigable layers**:

Layer	Content	Source
Structural	Containment hierarchy, modules, files, classes	Layer 1 (Axis E)
Execution	Control flow, call graphs, async structures	Layer 1 (Axes A–D)
Semantic	MeaningCells with `meaning` and `intention`	Layers 2–4
Boundary	Implied entities, external units, boundary calls	Layer 4

Navigation is the primary operation in SAM. Rendering is secondary. All access –
 whether by a
human exploring visually or an AI agent querying programmatically – is expressed
 through the
same three operations: graph selection, graph projection, and optional abstracti
on or expansion.

Five canonical **view types** are defined – Structural Hierarchy, Flow/Execution
, Variable/State,
Semantic Detail, and Boundary – each projecting a different cross-section of the
 same underlying model. A **stable rendering element identity** (`DiagramElement
Id`) ensures that interactive exploration, programmatic querying, and visual nav
igation can all operate on the same representation without ambiguity.

SAM deliberately provides **a single unified access model for both humans and AI
 agents**. There is no separate "human interface" and "agent API." The same grap
h selection and projection primitives serve both. This reflects the design princ
iple that the humans and AI systems collaborating on software must share not onl
y the knowledge base but also the conceptual framework for navigating it.