ThinkCapital LLC

Government IT / AI Governance Initiative

Functional Sizing as a Foundation for AI Governance Measurement

Applying Function Point Analysis and COSMIC to AI System Scope and Complexity

ThinkCapital GIAG Research Series Technical Methods Paper

March 2026

Section 1: The Wrong Unit Problem

Why measurement discipline matters for AI governance

The central argument of this paper is that AI governance frameworks are currently measuring the wrong things, and that the field of software measurement has already worked through an analogous problem with results that are directly applicable. Before making that case, it is worth being precise about what "applicable" means in this context, because the connection is not superficial.

Software functional sizing developed as a response to a specific governance failure: organizations could not compare, benchmark, or predict software project outcomes because the available metrics described implementation rather than function. Adoption rates, documentation scores, and compliance checklists in AI governance represent the same category of failure. They describe activity at the implementation layer of governance without reaching the functional layer where governance either works or does not work.

The goal of this section is to establish the measurement foundation that a credible AI governance framework needs. Function Point Analysis (FPA), as developed by Allan Albrecht at IBM[1] and validated by Capers Jones at Software Productivity Research,[2] provides a tested approach to sizing complex systems from the perspective of what they deliver to users rather than how they are built. Applied to AI governance, this approach gets part of the way to what is needed. It addresses the external behavior of AI systems, the transactions, data flows, and outputs that governance frameworks are designed to oversee.

What FPA does not reach, by design, is the internal computational logic of those systems. That gap is significant for AI governance because much of what makes AI systems risky or governable lives inside the application boundary, in the model behavior, the inference processes, and the decision pathways that produce outputs. The COSMIC functional sizing method, developed as an ISO standard for software systems where internal processing complexity is the primary scope driver,[5,6] extends measurement into that territory.

A hybrid approach combining IFPUG-style external boundary analysis with COSMIC-style internal process measurement offers a framework that accounts for both the functional surface of an AI system and the internal complexity that determines its behavior. That hybrid approach is the subject of active research under the Government IT and AI Governance Initiative (GIAG), and this section establishes the measurement logic that motivates it.

What functional sizing measures, and what it deliberately excludes

Functional sizing rests on a single organizing principle: a software system is measured by what it delivers to users, not by how it is built to deliver it. This requires drawing a conceptual line, the application boundary, around the system being measured. Once that boundary is established, the method counts only two categories of things: data transactions that cross the boundary in either direction, and data collections the application maintains on behalf of its users.

Everything inside the implementation layer, the code, the database engine, the algorithms, the infrastructure, the programming language, is invisible to the count by design. This is not a limitation. It is the methodology's central claim. Implementation details change with every technology cycle. What the software does for its users is stable across those changes, and it is the only thing that carries economic meaning for the organizations that commission and operate the system.

The five measurable component types that Albrecht defines fall cleanly into this framework:[14]

- External Inputs (EI) count distinct transactions that bring data into the system and update its internal records.
- External Outputs (EO) count transactions that derive and return processed results, reports, or summaries.
- External Inquiries (EQ) count retrieval transactions that return data without modifying any record.
- Internal Logical Files (ILF) count the distinct data collections the application maintains within its boundary.
- External Interface Files (EIF) count data collections maintained by another application that the measured system references.

Together, these five component types capture the complete functional surface of a system from the user's perspective. The exclusion zone is as important as the count zone. Source code and algorithms, database technology, operating systems, hardware, network configuration, and programming language are all explicitly outside the measurement. A transaction that requires three hundred lines of assembly to implement carries the same function point weight as the same transaction implemented in twenty lines of a higher-abstraction language. The technology choice is transparent to the measurement.

Figure 1. The Application Boundary: What Function Point Analysis Counts

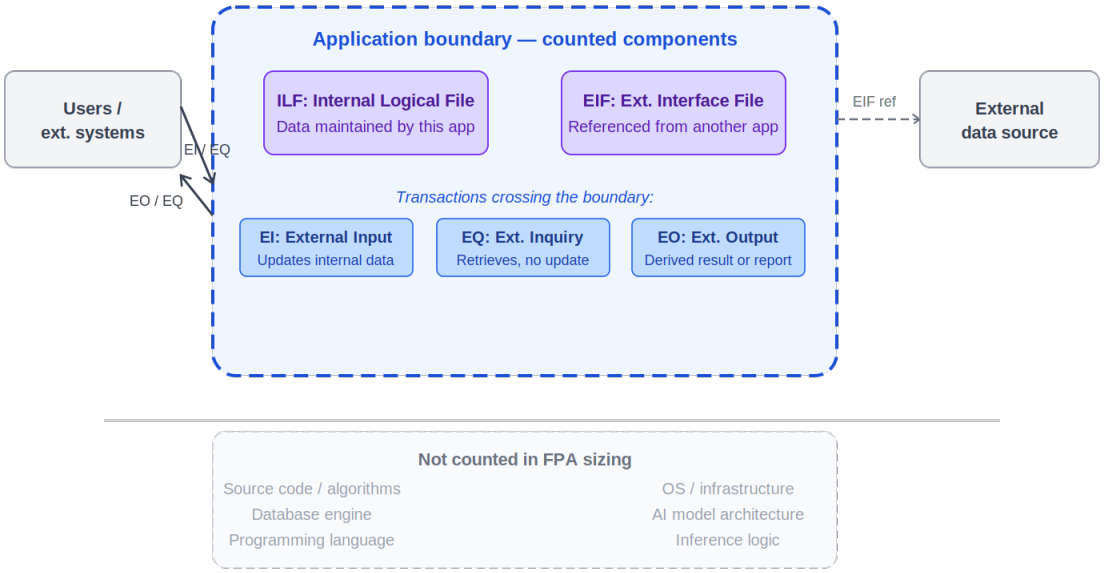

Figure 1. The Application Boundary: What Function Point Analysis Counts The dashed boundary separates counted functional components (above) from implementation elements that FPA deliberately excludes (below). Note that AI model architecture and inference logic appear in the exclusion zone (a gap that COSMIC addresses.)

From function point size to effort estimate

A raw function point count is the starting point for estimation, not the answer. The count establishes functional size. Effort is a separate derivation that applies the count through a structured chain of adjustments, and understanding that chain is what separates functional sizing as a measurement discipline from simpler artifact tallies.

Complexity weighting

Each component is rated on a three-level scale (simple, average, or complex) based on how many data element types it references and how many file types it interacts with. An external input that updates a single data field on a single file is simple. One that validates data against multiple reference tables and updates three internal logical files is complex. Each complexity level carries a different point weight.[4] This step converts the raw component count into an Unadjusted Function Point (UFP) total that reflects actual functional complexity rather than just component count.

Capability scores

The capability score is where the model becomes a management instrument rather than just a sizing exercise. The same function point count requires radically different effort depending on who is building the system and under what conditions. Jones's productivity model, built from the SPR database of enterprise assessments,[2] evaluates three dimensions: staff experience and skill in the relevant domain and technology, process maturity (how well-defined and consistently followed the development and quality practices are), and tool and environment quality.

The combination of these three scores establishes a productivity rate in hours per function point. This rate varies by a factor of three to five between a low-capability team and a high-capability team working on functionally identical requirements. A program manager using this framework understands why two projects of identical functional scope return different cost-per-deliverable figures. The answer is in the capability score, not in the project scope.

Constraint multipliers

Overlaid on the capability score are project-level constraint factors that amplify effort regardless of team capability. Schedule compression, forcing delivery ahead of the natural schedule for a project of that functional size, raises cost per function point because compressed schedules generate defects that must be found and fixed under deadline pressure. Requirements volatility is a second major multiplier: a project whose requirements change significantly during development pays a rework penalty that can double effective effort on the affected components. Technology novelty adds a learning-curve factor when teams are working in unfamiliar platforms or domains.

These are regression patterns drawn from the project database, not theoretical adjustments.[2] They are consistent relationships observed across projects of varying type, size, and domain. An agency procurement officer applying this model can translate scope-change events into effort and cost implications with a degree of precision that neither LOC counting nor checklist-based governance provides.

Lifecycle phase distribution and defect economics

The final step converts total estimated effort into a phased budget. On a well-run software project, effort is not concentrated in coding. It distributes across the full lifecycle, with testing consuming the largest share and requirements analysis consuming a critically important early share. The phase distribution matters beyond scheduling because defect removal cost escalates sharply across phases.

A requirements error caught during requirements review costs approximately one unit of effort to fix. The same error discovered during system testing costs twenty to fifty times more to

remediate. Caught in production, the cost multiplier reaches into the hundreds.[2] This means the lifecycle distribution of effort is a quality economics question, not just a scheduling question. Projects that under-invest in early phases systematically transfer a much larger liability to testing and production support.

Two features of this nonlinear model carry directly into the governance measurement argument. First, effort scales faster than proportionally with functional size because larger systems introduce coordination overhead, integration complexity, and defect propagation effects absent at smaller scale. Second, and more practically: a team that compresses requirements analysis by thirty percent does not save thirty percent of early-phase cost. It transfers a much larger liability forward.

Why LOC fails and why the functional boundary holds

The rejection of lines of code as the sizing unit is empirical and has a precise target. Jones documents what he calls the backfiring relationship: the number of source lines required to implement a single function point varies systematically by programming language.[3] Assembly requires approximately three hundred lines per function point. COBOL requires approximately one hundred. Object-oriented languages of the mainframe era required twenty-five to fifty. The specific figures are calibrated against technology environments that predate most current practice, but the underlying structural problem is not historical.

The backfiring relationship in modern languages and AI-generated code

The same structural disparity Jones documented persists across modern language choices. A transaction implemented in Java typically requires more lines than the same transaction in Python. Python requires more than an equivalent implementation in a TypeScript framework with mature library coverage. The ratio is smaller than the assembly-to-COBOL gap because generations of language abstraction have compressed the range, but it does not disappear.

What makes this newly urgent is AI-assisted code generation. When a developer using GitHub Copilot, Cursor, or a comparable tool generates a function from a natural language prompt, the LOC count reflects the tool's generation style as much as the task's functional complexity. Two developers working on functionally identical requirements, one generating code manually in Python and one generating it through an AI assistant in a more verbose style, produce different LOC counts for the same deliverable. The metric becomes a measure of the generation tool's verbosity preferences.

Function points hold through this transition without modification. The functional requirement is the same regardless of how the code was generated. The stability of functional sizing across implementation method is precisely the property that makes it viable for benchmarking in an environment where code generation practices are changing faster than any fixed LOC-per-function-point table can track.

More damaging than the cross-language comparison problem is what LOC measurement does inside a single technology environment. A developer who writes cleaner, more efficient code in fewer lines registers as less productive than a peer who writes verbose, redundant code producing the same functional output. The metric rewards the behavior it should discourage. Jones is explicit that this is not a corner case. It is the predictable consequence of using a metric that measures implementation volume rather than functional delivery.[3]

The defect dimension of AI-generated code reinforces the same conclusion from a different direction. Emerging research on AI-assisted development suggests that well-prompted generation in bounded, well-specified functional contexts produces code with lower initial defect density than manually written code of equivalent functional scope.[7] Fewer implementation-layer defects arrive at the testing phase. But specification errors, edge case gaps, and behavioral misalignments between generated code and actual business requirements remain, and in some cases become harder to detect precisely because the code surface looks clean. The lifecycle defect cost model Jones established holds. What shifts is where in the lifecycle the defects concentrate.

Function points hold because the application boundary severs the connection between implementation choice and functional measurement. A case management system implemented in Java carries the same function point count as the same system reimplemented in Python or generated through an AI-assisted toolchain. The technology and generation method decisions are transparent to the measurement. That transparency is the design requirement Albrecht set out to satisfy, and it is what makes functional sizing viable for cross-portfolio and cross-organization benchmarking in an era of rapidly shifting development practice.

What function points cannot reach: the case for COSMIC

The application boundary that gives FPA its stability also defines its limit. By design, the IFPUG method counts what crosses the boundary and what is stored within it.[4] It does not count what happens inside the system to produce those outputs. For most enterprise software, transaction-oriented systems processing well-defined business rules against structured data, this is acceptable. The functional complexity is largely expressed at the boundary, and internal processing is relatively bounded and inspectable.

Modern AI systems break that assumption in a specific and consequential way. The outputs of an AI system (a risk classification, a procurement recommendation, a document summary, a fraud detection flag) cross the application boundary in a form that FPA can count and classify. But the internal behavior that produces those outputs is where governance exposure lives. The model architecture, the training data distribution, the inference pathway, the confidence

calibration, the handling of inputs that fall outside the training distribution: none of these appear in an FPA count.

Consider a practical contrast relevant to federal agency deployment. Two systems both produce a case prioritization score as an external output. The first evaluates five criteria against a lookup table and applies a weighted sum. The second is a transformer-based classification model trained on historical case outcomes. Both systems return the same output type to the same users. An FPA count rates them identically on the external output component. A governance framework that treats them identically is missing the entire risk surface of the second system: the opacity of the inference pathway, the sensitivity to input perturbations, and the potential for the training distribution to reflect historical patterns that should not be perpetuated in current decisions.

COSMIC (Common Software Measurement International Consortium) functional sizing was developed to address software systems where internal processing complexity is the primary driver of scope and behavior.[5, 6] Originally built for real-time, embedded, and infrastructure software, COSMIC measures data movements within the application boundary rather than only at the boundary surface. A COSMIC measurement identifies functional processes (discrete units of software behavior each triggered by a distinct data movement event) and counts the data movements that constitute each process: entries from the triggering layer, exits returning results, reads from persistent storage, and writes to persistent storage.

Applied to an AI inference pipeline, COSMIC-style internal process measurement provides a basis for characterizing the computational pathway from input to output. A retrieval-augmented generation (RAG) system that takes a user query, retrieves relevant document chunks from a vector store, constructs a context window, passes it through a large language model, and returns a synthesized response involves a distinct and countable sequence of internal data movements. Each retrieval step is a read. Each document chunk assembled into context is an entry to the synthesis process. The generated response returned to the user is an exit. The COSMIC count across this pathway is larger and more differentiated than the FPA count at the boundary, and that differentiation is what governance-relevant analysis requires.

Figure 2. COSMIC Extends Measurement Inside the Application Boundary

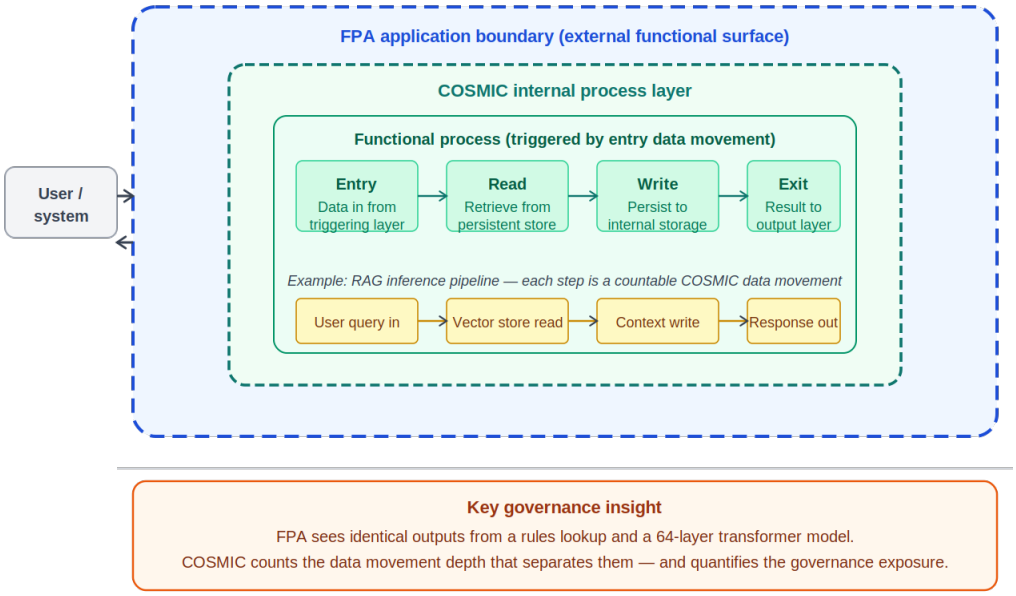

Figure 2. COSMIC Extends Measurement Inside the Application Boundary FPA sees only what crosses the boundary. COSMIC counts the data movement steps within each functional process, making the computational depth of AI inference pipelines visible and quantifiable. The governance insight illustrates why this distinction matters: FPA cannot differentiate a rules-based lookup from a large language model inference pipeline that produces the same external output type.

A low COSMIC count for an AI process, relative to its FPA-measured external complexity, suggests a simple, inspectable inference pathway. A high COSMIC count for a system of similar external complexity suggests a deep, multi-stage process where the relationship between input and output is harder to trace and harder to audit. Neither characterization is inherently good or bad. Both are operationally relevant to oversight: how much internal processing stands between a user request and an agency decision output, how many data sources are consulted in that pathway, and how stable the pathway is across inputs that vary in type or quality.

The hybrid approach and the current research agenda

Neither FPA nor COSMIC alone provides the complete measurement framework that AI governance requires. FPA establishes the functional surface: what the system accepts, what it produces, what data it maintains, and how those elements relate to user-visible behavior. COSMIC extends into the internal processing layer where AI system behavior and risk actually originate.

A hybrid approach combining IFPUG-style boundary analysis with COSMIC-style internal process measurement represents the next logical step, and it is the framework currently under

development and validation through the Government IT and AI Governance Initiative research streams. The external boundary analysis produces a functional size baseline that allows cross-agency and cross-system comparison on scope and external complexity. The internal process measurement adds a computational pathway characterization that supports governance-relevant differentiation between systems that look similar from the outside but operate very differently inside.

Figure 3. Hybrid FPA + COSMIC Framework for AI Governance Measurement

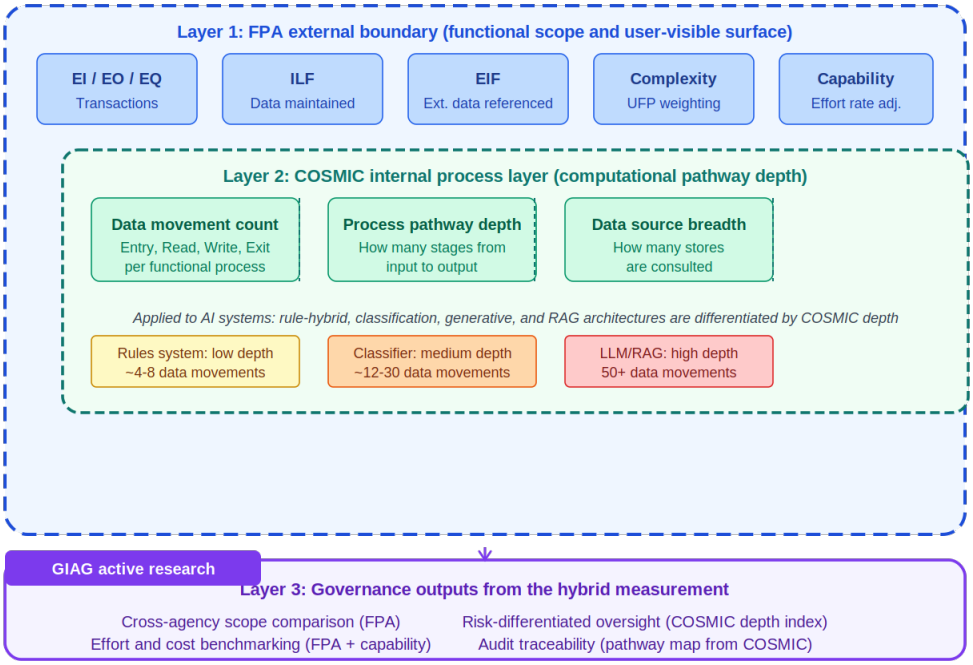

Figure 3. Hybrid FPA + COSMIC Framework for AI Governance Measurement Layer 1 (FPA) establishes functional scope and the external surface, enabling cross-agency benchmarking and effort estimation. Layer 2 (COSMIC) characterizes internal process depth, differentiating systems by computational complexity and audit tractability. Layer 3 represents the governance outputs the combined measurement produces. The research badge indicates that the practical methodology for Layer 2 as applied to probabilistic AI systems is under active development through GIAG.

The practical research question is whether the COSMIC data movement taxonomy, developed for deterministic software, translates cleanly to probabilistic AI systems where the same input does not reliably produce the same sequence of internal operations. There is theoretical and preliminary empirical reason to believe it does, particularly for the class of AI systems most common in government deployment: retrieval-augmented pipelines, classification models with fixed inference graphs, and rule-hybrid systems that combine deterministic logic with model-based components. For generative systems with fully dynamic inference paths, the mapping requires additional methodological development, and that is what the current research is designed to produce.

The governance measurement problem in AI is not structurally new. It is the same problem Jones and Albrecht solved for enterprise software: replace a metric that measures implementation artifacts with a metric that measures functional and behavioral reality. The tools have a forty-year intellectual lineage. The adaptation work, extending them to cover internal process complexity in AI systems, is the current frontier. The remainder of this paper establishes why that frontier matters by demonstrating where current AI governance metrics fail at the functional layer.

References

[1] Albrecht, A.J. (1979). "Measuring Application Development Productivity." *Proceedings of the IBM Applications Development Symposium, SHARE/GUIDE,* Monterey, CA. IBM Corporation.

[2] Jones, C. (2008). *Applied Software Measurement: Global Analysis of Productivity and Quality,* 3rd ed. McGraw-Hill.

[3] Jones, C. (1986). *Programming Productivity.* McGraw-Hill. (Primary source for the backfiring relationship and LOC-to-function-point conversion tables.)

[4] International Function Point Users Group (IFPUG). (2010). *Function Point Counting Practices Manual, Release 4.3.1.* IFPUG. Available at ifpug.org.

[5] COSMIC. (2021). *The COSMIC Functional Size Measurement Method, Version 4.0.2.* Common Software Measurement International Consortium. Available at cosmicon.com.

[6] ISO/IEC 19761:2011. *Software Engineering — COSMIC: A Functional Size Measurement Method.* International Organization for Standardization, Geneva.

[7] Ziegler, A., Goldmann, L., de Vries, G., Sobania, D., et al. (2022). "Productivity Assessment of Neural Code Completion." *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE 2022),* ACM, New York. doi:10.1145/3540250.3558965. (Microsoft Research affiliation; peer-reviewed study of AI code completion effects on developer productivity and defect rates.)