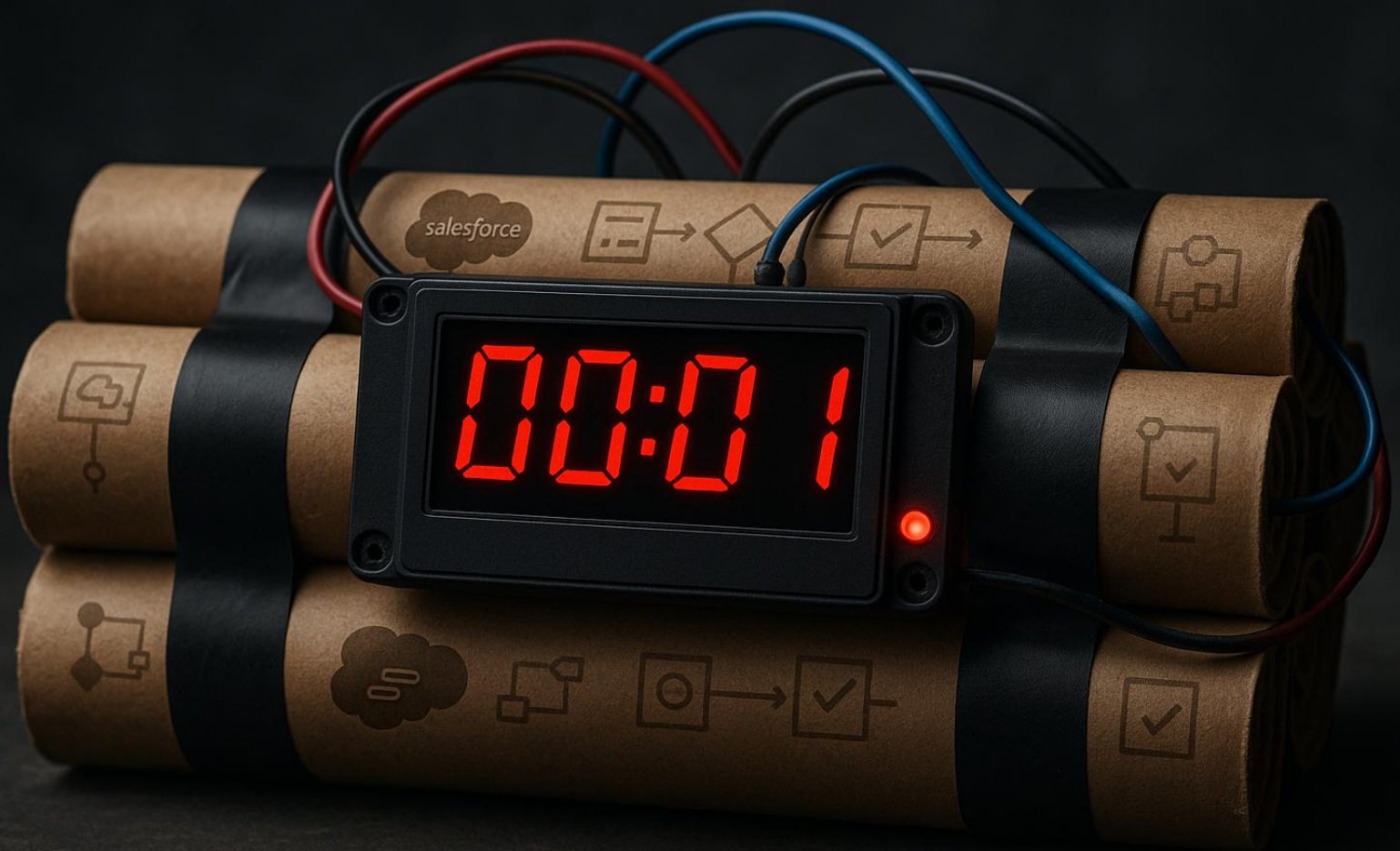




White Paper

Entropy: The Hidden Time Bomb in Your Org

How Observability Combats Inevitable
Software Rot in Salesforce and Other Systems



Executive Summary

Every software system, including your Salesforce org, starts out in good health—clean, efficient, and built for business agility.

But over time, something insidious creeps in. Some call it software rot. More formally, it's called "entropy," a term borrowed from science. What makes entropy so dangerous – and worth paying attention to – is that the rot occurs *inevitably* as layers of Apex code, Flows, managed packages, integrations, and quick fixes accumulate. Eventually, even simple changes feel risky. What began as a strategic platform degrades into a brittle puzzle.

In large enterprises, the problem compounds with scale. Multiple teams, conflicting priorities, and years of legacy logic make entropy more than a nuisance—it becomes a threat to system stability, innovation, and uptime.

This white paper explores how to fight back. With a strategy grounded in runtime observability, you can pinpoint hotspots and patterns of decay. You'll see how your org is really behaving and evolving. With solid data in hand, you'll be able to advance beyond troubleshooting and get to a state where you and your team can proactively fight back the creeping chaos that threatens every mature Salesforce environment.

June 2025

Introduction

Modern software systems exhibit a tendency to become more complex and disordered as they grow and evolve. This increase in “entropy” – a term borrowed from thermodynamics – manifests as tangled code, proliferating dependencies, and unpredictable behavior in the software.

Without active management, an evolving codebase or configuration will accumulate **technical debt** and complexity, making it harder to maintain over time [1, 2]. Enterprise applications and platforms (such as Salesforce) are especially prone to this entropy growth as new features, customizations, and integrations are layered on year after year. This whitepaper explores software entropy and how it grows. It will then show you how the practice of **observability** – the ability to closely monitor and understand system internals – can be a crucial strategy for keeping complexity manageable in large and growing Salesforce orgs.

We'll begin by examining academic research on **entropy in software systems** and how complexity tends to increase over time. Next, we'll look at how observability principles can counteract entropy by improving insight into system behavior. We then apply these concepts to hypothetical Salesforce orgs of increasing size (from a small org with a handful of customizations to a very large org with thousands of components), illustrating how entropy grows as the org becomes more complex. Finally, we'll examine why rising entropy in a growing Salesforce org is inevitable and eventually become a risk for the business running it—but also how a strong observability practice can make this complexity controllable.

Entropy in Evolving Software Systems

Effective refactoring, redesign, and modularization—the classical counter-measures to rising complexity—only work when teams **first know exactly where disorder resides and when restructuring will generate the greatest return**. In other words, *visibility* is the prerequisite for reducing entropy, not the other way around.

In software engineering, *entropy* is the metaphor for the growing disorder of a system as it changes over time [3]. Jacobson, *et al.*, put it bluntly: “As a system is modified, its disorder, or entropy, tends to increase” [4]. Every enhancement or hot-fix introduces a

small deviation from the original design; unless those deviations are tamed, they accumulate. Lehman's Second Law of software evolution formalizes this:

"As an E-type system evolves, its complexity increases **unless work is done** to maintain or reduce it." [5]

Left unmanaged, complexity degrades structure and maintainability, trapping developers in a vicious cycle of fragile code and escalating defects. [6, 7]. High entropy codebases become difficult for developers to understand, leading to more mistakes and further deterioration—a vicious cycle [8].

What Large-Scale GitHub Data Tells Us

Researchers have tried to measure this drift toward disorder. Hassan and Holt's **source-code change entropy** [9] applies Shannon's formula to commit history: when a single change scatters across many unrelated areas, the process is chaotic. Their studies (and later replications) show a strong correlation between high change-entropy and future defects.

A broader view comes from a **study of ~10,000 GitHub repositories** [10]:

- **Growth:** Codebases doubled in size every 3-4 years ($\approx 21\%$ compound annual growth in LOC) [11].
- **Community engagement:** Figure 1 (from the original dataset) plots entropy over project age, color-coded by open issues and forks. Active, long-lived projects cluster near the origin, showing that sustained investment in refactoring correlates with both lower entropy and healthier communities.

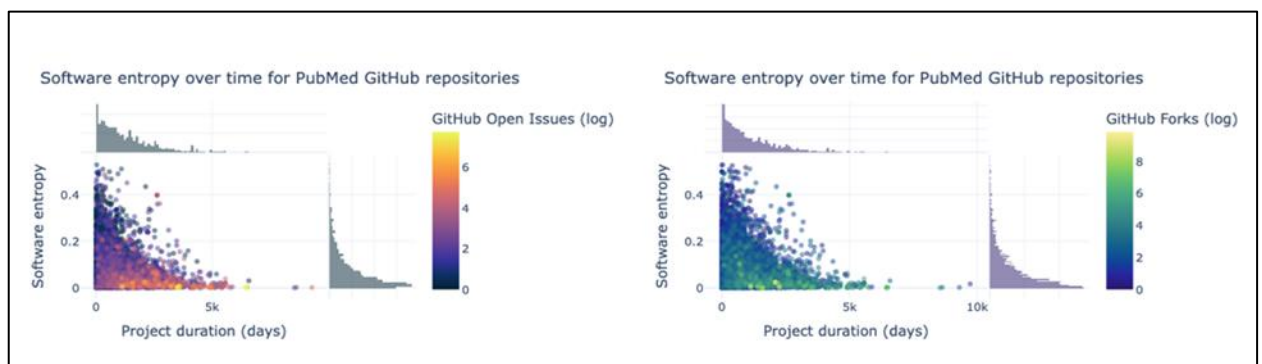


Figure 1

- **Defect prediction:** High-entropy “hotspots” aligned with modules that accumulated bugs fastest.
- **Entropy vs. longevity:** Projects with shorter lifespans exhibited significantly higher normalized entropy throughout their history, while long-lived projects kept entropy near the baseline through continuous cleanup [12].

These findings give empirical teeth to Lehman’s law:

“Without deliberate action, complexity rises; with active stewardship, entropy can be contained.

Why Salesforce Orgs Drift Faster

Everything documented above for open-source codebases is magnified in an enterprise SaaS environment like Salesforce:

- **Scattered, dependency-rich work.** Custom objects, flows, triggers, managed packages, and integrations cross-reference one another, so a “small” feature can ripple through dozens of metadata components.
- **Short project and tenure horizons.** Consulting engagements routinely last 6–12 weeks, and in-house dev/admin tenure averages 18–24 months; teams often move on before circling back for cleanup.
- **Competing business priorities.** Admins and developers juggle support tickets, sprint deadlines, and stakeholder requests; refactoring time is the first to be cut.
- **Multiple parallel teams.** Sales, Support, Marketing, and external vendors can all deploy to the same org, producing isolated “islands of logic” that nobody owns end-to-end.

The net result is **accelerated entropy growth**: configuration and code sprawl faster than in a single-purpose GitHub repository, and the window for structured cleanup is dramatically shorter. Without precise insight into which parts of the org are decaying, teams patch symptoms and move on — perpetuating the cycle.

Notably, Lehman's law never claimed that soaring complexity is inevitable—only that it is inevitable **unless we act**. For Salesforce orgs, acting starts with an **observability-first** mindset:

- **Objective runtime data, not gut feel.** Static code scans tell you what might happen; logs, metrics, and traces reveal what is happening. By capturing execution paths, governor limits, SOQL patterns, and metadata dependencies in real time, observability turns vague hunches about “messy areas” into quantifiable hotspots with hard numbers behind them.
- **Continuous measurement of entropy.** When every transaction emits structured telemetry, you can compute change-entropy on live traffic, watch it rise or fall after each release, and prove whether a refactor actually paid off. Entropy becomes an operational metric, no longer an abstract concept.
- **Actionable insight at the right granularity.** Because telemetry is tied to specific flows, Apex classes, and configurations, teams can isolate the smallest viable surface for cleanup, minimizing risk and business disruption.
- **Economic framing for stakeholders.** By showing how hotspots correlate with incident counts, MTTR, or page-load latency, observability exposes the real, tangible cost of unchecked complexity—language that resonates with executives holding the budget for technical debt pay-down.

Equipped with this objective lens, teams can schedule refactors when the data shows the benefit will outweigh the disruption, automate alerts when entropy exceeds agreed-upon thresholds, and track progress toward a healthier org with the same discipline they apply to revenue or uptime. The next section explores how modern observability platforms deliver these capabilities at enterprise scale, making sustainable entropy control a repeatable practice rather than a one-off crusade.

Observability as a Countermeasure to Complexity and Entropy

Observability in software systems refers to the ability to monitor and understand the internal state of the system based on the information it exposes – typically through logs, metrics, traces, and other instrumentation. In essence, an observable system is one that allows engineers to ask questions about its behavior (“Why is this request slow?”, “What happens when I trigger this process?”) and get insightful answers from the data. While observability is often discussed in the context of reliability and performance, it is equally critical for managing complexity and entropy in large systems. By shining light into the system’s inner workings, observability helps teams detect where disorder is accumulating and how different parts of the system interrelate.

At a high level, **observability makes complexity visible**. Instead of a black-box system where entropy can lurk unnoticed, an observable system provides continuous feedback about its operations. This has several benefits for controlling entropy:

- **Understanding Dependencies and Interactions:** Complex software (e.g. microservices architectures or highly customized platforms) may have many components interacting in non-obvious ways. Observability is “vital in managing system complexity by [providing] a profound understanding of the dependencies and interactions among components”. For example, distributed tracing can show how a single transaction weaves through dozens of microservices, revealing hidden couplings. In a Salesforce context, observability might mean having detailed logs or graphs of which processes (flows, triggers) fired in what order and how data flowed through them. This granular insight makes it easier to handle a complex system, because engineers can map cause and effect even amidst a tangle of connections.
- **Early Identification of Anomalies and “Entropy Hotspots”:** As systems grow chaotic, you often see symptoms – erratic performance, frequent errors in certain modules, data integrity issues, etc. Robust observability (with alerting, dashboards, anomaly detection) brings these signals to the surface early. It enables teams to pinpoint problematic areas in the code or configuration that are becoming overly complex or fragile. In other words, observability can highlight areas of high entropy so they can be refactored or stabilized proactively. Real-world practices like Netflix’s Chaos Engineering are predicated on observability:

by intentionally introducing turbulent conditions and monitoring the system's response, teams can discover weaknesses in the complexity and fix them before they cause outages.

- **Improving Maintainability via Transparency:** A highly complex system is manageable only if the people working on it can understand what's going on. Observability provides the tooling and data needed to understand even intricate behaviors. One industry article notes that unlike basic monitoring, observability goes deeper by "uncovering hidden issues and offering detailed insights into system dynamics, empowering developers to manage and troubleshoot application intricacies effectively." When developers trust that they can observe any part of the system's behavior in real time (such as seeing detailed debug logs or execution traces in a Salesforce org), they are more equipped to tame complexity. It reduces the fear of the unknown. Instead of ad-hoc guesswork in a messy system, engineers rely on evidence and data, which leads to better decisions about simplifying or correcting behavior.

Importantly, observability is a practice and design principle, not a specific product. It involves instrumenting your system to expose state, establishing good logging and monitoring conventions, and continuously analyzing the outputs. In the context of entropy, we can think of observability as providing **negative feedback** in the system evolution process. While entropy growth is like a natural "positive feedback" (complexity begets more complexity if unchecked), observability injects a balancing force: it allows the team to see the complexity and apply corrective action (refactoring, optimization) in a targeted way. A simple analogy: if a codebase is a garden that can become overgrown (weeds = technical debt, tangled vines = dependencies), observability is like good gardening tools and regular inspections – you spot the overgrowth and prune it. As one data engineering expert put it, because *"Data entropy represents disorder and chaos,"* **data observability solutions emerged specifically to "solve this type of problem"** by making health and usage of the system transparent. The same goes for software systems: observability brings order to chaos by enabling understanding.

In summary, observability helps reduce software entropy by illuminating the system's inner workings, thus aiding in complexity management. An observable system is easier to debug and improve, which means teams can refactor with confidence. They can also establish baselines and metrics to **quantify complexity** (e.g., track the number of inter-

module calls, or monitor how many deprecated fields are still in use) and then watch those metrics as they make changes to reduce entropy. By continuously observing, the team effectively closes the loop and keeps entropy in check instead of letting it spiral uncontrolled.

Complexity and Entropy in Growing Salesforce Orgs

Large Salesforce orgs provide a prime example of entropy accumulation in a configuration-heavy software system. A Salesforce org starts as a relatively simple environment – perhaps a dozen custom objects, a few automation scripts, and a small group of users. Over time, as the business expands and new requirements emerge, the org grows: more custom objects and fields are added, more Apex classes (custom code) are written, more Flows and Process Builders (automation) are configured, integrations bring in new data, and so on. Each of these changes individually seems harmless, but together they dramatically increase the overall complexity of the org’s metadata. Admins and developers often notice that an org which was nimble and easy to change in its first year becomes much harder to manage a few years later – this is entropy rising.

Before we model how entropy accelerates as an org scales, it helps to explain **why this paper spotlights three metadata families—custom objects, Flows, and Apex classes**:

- **Custom objects** anchor the data model. Each new object usually spawns new relationships, record types, page layouts, and security rules, all of which expand the surface area developers and admins must reason about.
- **Flows and Apex** are the primary engines of business-process automation. When data objects multiply, so do the scripts and declarative automations that act on them, creating a dense web of dependencies.
- **A conservative lens.** Salesforce offers hundreds of additional configuration points (validation rules, formula fields, Lightning pages, permission-set groups, etc.). Limiting our analysis to objects, Flows, and Apex therefore understates the true scope of customization and keeps the “entropy baseline” intentionally conservative.

With that framing in mind, let’s perform a concrete thought experiment. Consider **ten hypothetical Salesforce orgs** (called Org 1 through Org 10) representing a spectrum from a small implementation to an extremely large, enterprise implementation. Org 1 might have on the order of *20 custom objects, 10 Apex classes, 5 Flows, and 100 users*. At the other extreme, Org 10 could have *500+ custom objects, 600+ Apex classes, 200+ Flows, and 10,000 users*. Table 1 below outlines one possible growth trajectory:

Hypothetical growth of a Salesforce org from small to very large scale

This table illustrates the orders-of-magnitude increases in metadata components. (These figures are for illustration; actual orgs vary, but many enterprises do reach the upper end – having hundreds of custom objects and classes – after years of growth.)

Org	Custom Objects	Apex Classes	Flows	Users
Org 1 (small)	20 objects	10 classes	5 Flows	100 users
Org 2	30 objects	16 classes	8 Flows	150 users
Org 3	45 objects	25 classes	12 Flows	280 users
Org 4	62 objects	40 classes	18 Flows	460 users
Org 5	91 objects	64 classes	27 Flows	770 users
Org 6	132 objects	102 classes	41 Flows	1,290 users
Org 7	193 objects	162 classes	62 Flows	2,150 users
Org 8	282 objects	257 classes	95 Flows	3,590 users
Org 9	411 objects	409 classes	144 Flows	6,000 users
Org 10 (large)	600 objects	650 classes	220 Flows	10,000 users

Table 1

With each step up in org size, **new types of complexity emerge**.

In Org 1, a single developer-admin can mentally keep track of how things are connected. But by Org 5 or Org 6, no single person grasps the full picture – the org now contains dozens of interconnected objects, automations, and code modules.

By Org 10, the system is incredibly intricate: if not managed well, it may be what one study called “unmanageably complex.” Let’s break down the kinds of dependencies and interdependencies that accumulate in such a Salesforce org as entropy grows:

- **Object Relationships:** Salesforce is highly data-driven, so custom objects often reference each other via lookup or master-detail relationships. In a small org, you might have one or two key relationships (e.g., a **Project** object linking to **Account**). In a large org with 500+ objects, if on average each object has a couple of relationships, that’s on the order of a thousand inter-object links criss-crossing the data model. The schema becomes a dense network. This means a change to one object (say, renaming or altering a field that other objects look up to) can have ripple effects across many others.
- **Field Usage and Formulas:** As objects multiply, so do custom fields – large orgs can have tens of thousands of fields. Many fields contain formulas or validation rules that reference fields on other objects. For example, an **Opportunity** might have a formula that pulls data from its related **Account** or **Product** records. Each such formula or rule is another dependency line in the web. High entropy here manifests as difficulty in knowing which fields depend on which others – a seemingly simple change like altering a field’s data type might break several formula fields or downstream integrations. (In fact, orgs often accumulate “dead” fields that are no longer used but are still hanging around, adding to clutter and confusion.)
- **Automation (Flows, Process Builders, Triggers):** Salesforce provides declarative automation (Workflow Rules, Process Builder, Flows) and programmatic triggers (Apex triggers) that respond to data changes. In a small org, you might have one Flow on a key object. In a complex org, every major object could have multiple Flows or a combination of Flows and Apex triggers, sometimes layered over the years. These automations can call each other (e.g., a record update by a Flow fires a trigger on that object, which in turn calls some Apex). Without careful design, you get **automation entropy**: tangled logic

spread across many Flow diagrams and trigger scripts. It becomes non-trivial to predict the order of execution or the combined effect when a record is saved, because so many rules kick in. This is a classic entropy symptom – **loss of predictability**.

- **Apex Classes and Integrations:** Custom Apex code brings power and also complexity. Each Apex class might implement business logic involving multiple objects. Classes can call other classes (like a utility library or a service layer), forming a call graph of dependencies similar to a traditional codebase. In a large org with hundreds of classes, you often find tightly coupled code modules – e.g., an “Order Management” class that invokes methods from “Inventory Service” and “Finance API” classes, which in turn query various objects. When requirements change, making an update in such a codebase is risky because any given class may interact with many others. The more interdependent the classes, the higher the entropy of the code structure. If the org integrates with external systems (via API calls), those external dependencies increase complexity further (though those are outside the metadata count, they add to the cognitive load and points of failure).

All these factors compound as the org grows. Notably, the growth in *number of components* (objects, fields, flows, classes, etc.) often leads to an even faster growth in *number of connections* among components. If we treat each metadata element as a node in a graph and each dependency (reference, invocation, relationship) as an edge, the graph becomes more densely connected as it gets larger.

In our hypothetical series from Org 1 to Org 10, even if each new object or automation only brings a relatively fixed number of new dependencies, the *total* number of dependency links can increase super-linearly. This is analogous to a network effect – the “graph” of the org’s design explodes in complexity.

Growth of Components and Dependencies in Salesforce Orgs

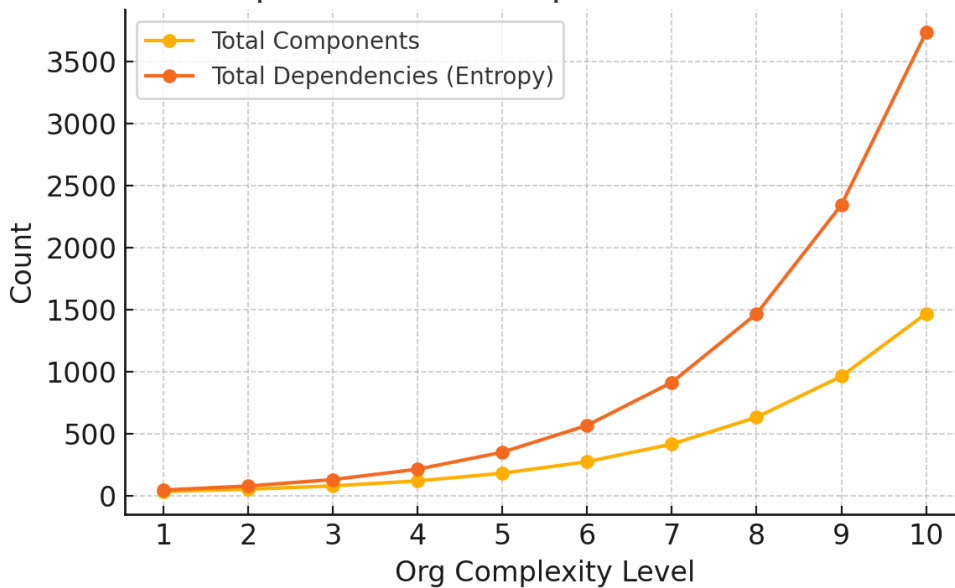


Figure 2

This graph (Figure 2) shows the growth of metadata components versus interdependencies (a proxy for entropy) across ten hypothetical Salesforce orgs (small to large). The **yellow line** shows the total count of *components* (custom objects, Apex classes, Flows), increasing from a few dozen to over a thousand as the org scales up. The **orange line** shows the potential number of *dependency links* (object relationships, cross-references in code/flows, etc.), which grows even more rapidly – in this model approaching exponential growth.

By Org 10, the number of interdependent connections is on the order of several thousand. This illustrates how entropy (disorder due to many intertwined parts) accelerates with system size. In fact, adding components at a steady linear rate can cause exponential increases in complexity due to the multiplicative effect of interdependencies.

As Figure 2 suggests, **entropy in a Salesforce org can be measured by the proliferation of dependencies and the difficulty of untangling them.**

It's not just the 600 objects or 650 classes alone that make Org 10 complex – it's how they are connected. If those 600 objects were all completely isolated (which of course they aren't), the system would actually be easier to reason about. It is the entanglement – e.g., a web where Object A's change triggers a Flow that updates Object B, which

invokes an Apex class that queries Object C – that creates an entropic system where a small change may have unpredictable consequences.

Indeed, a Salesforce whitepaper by Metazoa [13] notes that even seemingly simple growth (like adding profiles, layouts, or objects linearly) can lead to an **exponential rise in user permission combinations and other entangled assets (fields, record types, Apex classes)**, ultimately increasing overall org complexity at “an even higher rate.” In large, long-lived orgs, *“many Salesforce Orgs become more complex every day.”* [13]

This entropy manifests in practical pain points for Salesforce teams: for example, an admin might be afraid to delete a field because they’re not sure where it’s used; or a developer might hesitate to modify a Flow because that Flow’s effect on downstream processes isn’t clear. Over time, the fear of breaking something leads to inaction, and the old artifacts (fields, flows, code) pile up as “brittle scaffolding.” It’s easy to see how an older org can accumulate hundreds of unused or redundant components (duplicate fields, outdated automations) that no one removed because the dependencies weren’t understood. That is entropy in action.



Entropy can be a major source of pain for Salesforce teams, but it can be managed with system visibility.

However, just as with general software entropy, this complexity in Salesforce orgs can be managed—with the right approach. The key is to regain visibility into the system.

Making Entropy Manageable with Observability in Salesforce

By now, it’s clear that a growing Salesforce org will inevitably become complex. Complexity is the byproduct of success – the more business processes you support on the platform, the more moving parts you’ll have. In fact, complexity in such systems is **inevitable** and even, “not bad” in itself, says Salma Bakouk, founder and CEO of Sifflet Data. Instead, she notes that it’s the sign of a richly featured system. “The key here is to manage complexity to make the system easier to understand and work with.” [14] This is where applying observability concepts can make a big difference.

The Strategic Counterweight to Org Entropy: Runtime-Centric Observability

Managing — not eliminating — that moving complexity is the goal, and modern observability practices give teams the data to do it methodically.

A **runtime-first observability stack** focuses on what the platform is actually doing:

RUNTIME SIGNAL	WHY IT MATTERS FOR ENTROPY	TYPICAL ARTIFACT
Transaction traces & debug logs	Pinpoint which automations execute, in what order, and how long they consume CPU/DB time. Hotspots surface immediately.	Apex/Flow debug logs, Event Monitoring, APM traces
Structured error & governor-limit events	Highlight fragile areas where limits are breached or exceptions erupt, guiding targeted stabilization.	Custom Error Log object, EventLogFile
Usage analytics	Quantify which parts of the system, objects, fields, Flows, and reports are truly used vs. dormant “dead weight,” letting teams cut unused complexity safely.	Field-usage stats, Flow invocation counts
Performance baselines & SLOs	Provide an impartial KPI for system health; teams can watch entropy creep as latency or error budgets erode.	Custom dashboards, DataDog/New Relic SLO widgets

Table 2

Why not lean on hypothetical impact analyses?

Dependency graphs and change-impact simulations remain useful auxiliaries — they tell you what might break. But runtime observability proves what is breaking (or about to), forcing data-driven decisions instead of educated guesses.

Briefly, those auxiliary views:

- **Dependency visualization** gives a quick map (“Field X touches 3 formulas, 2 classes, 1 Flow”) so you’re not flying blind during a change, but it doesn’t reveal which paths are *active* or *slow*.

- **Change-impact assessment models** prospective risks in a sandbox; runtime telemetry validates or contradicts those models once the change lands.

Taken together, runtime observability:

1. **Locates the true hotspots**—the 20 % of automations causing 80 % of incidents or governor-limit burn.
2. **Sets an objective entropy KPI**—latency, error rate, or change-entropy scores that trend up when complexity grows unchecked.
3. **Enforces data-driven stewardship**—clean-up sprints target the worst offenders first, and refactor wins are verified by post-release metrics instead of anecdote.

In short, observability turns the fight against Salesforce org entropy from whack-a-mole reactions into a strategic, measurable, and repeatable engineering discipline.

Observability brings **transparency** and **data-driven management** to a complex Salesforce org. Instead of being intimidated by an opaque mass of metadata, the team can systematically analyze and monitor the org's internals. This enables ongoing maintenance practices that fight entropy: for example, instituting regular clean-up sprints informed by metrics (delete or consolidate things that logs show aren't used), or setting up alerts when an unusually high number of errors start coming from a certain Apex class (indicating maybe that area of the system is becoming problematic as new features get bolted on).

Embracing Entropy, Salesforce-Style

GitHub-scale research shows that entropy rises predictably as projects add code and contributors. In a Salesforce org, those same forces are **multiplied**: each new **custom object** seeds page layouts, validation rules, and record types; every **Flow** or **Apex class** stitched on top of it spawns more cross-references, limits checks, and governor-budget contention. Add in declarative features—formula fields, Lightning pages, permission-set groups—and the combinatorial surface outstrips any typical Git repository. Entropy is therefore **unavoidable**, but it is **not unmanageable**.

Why Runtime-Level Observability Is the Control Valve

1. **Real usage beats hypothetical impact.**

A dependency graph can warn you that **Opportunity.Stage** touches five Flows and two Apex triggers. Only runtime telemetry—debug logs, Event Monitoring, transaction traces—can confirm whether those paths are firing 10,000 times a day or are lying dormant. Data, not speculation, decides what to refactor first.

2. **Entropy becomes a measurable KPI.**

When every transaction emits structured events (CPU time, SOQL counts, stack traces), you can compute a running org-entropy score or hotspot leaderboard. Spikes after a release show complexity is outpacing control; a post-refactor drop proves the cleanup worked.

3. **Targeted clean-up, minimal blast radius.**

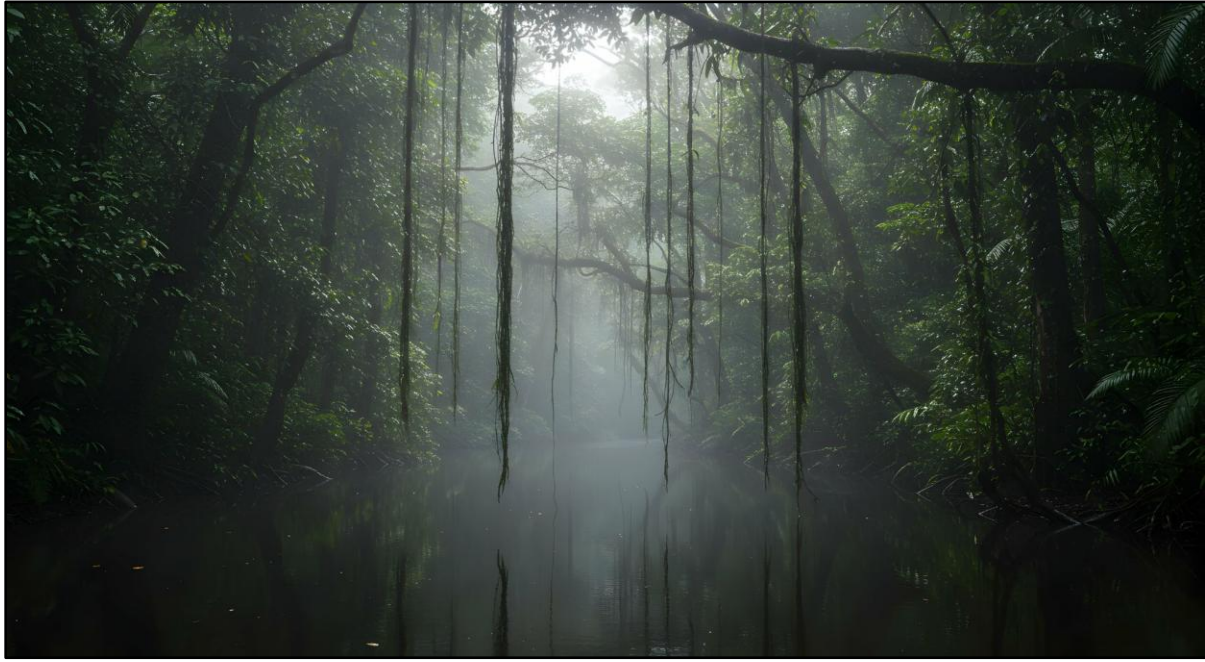
Suppose traces reveal that saving a Contact triggers 12 automations and burns 80 % of the CPU limit. Teams can merge redundant Flows or shift logic into a single orchestrator, then watch CPU time fall in the next deploy—objective proof that entropy was improved where it mattered.

4. **Shared, impartial language for stakeholders.**

Dashboards that correlate hotspot entropy with incident counts or page-load latency help translate “technical debt” into metrics decision makers care about, like dollars of downtime or hours of admin toil. Expressed in these business-case terms, it’s much easier to show impact and unlock budget for follow-up work.

Key Takeaways: Is Your Org a Jungle or a Managed Garden?

Nothing in the universe is static, and the same is true of your org. Even if your Salesforce environment is stable and trouble-free now, you'll have to work to keep it that way.



Your Org on Entropy

Whether your org becomes a jungle (where unfortunate events lurk in the shadows), or a well-manicured garden (where anomalies stand out and can be quickly addressed) depends on how proactive your org health strategy is.

Treat the org as a living, observable system and take these actions:

- **Collect runtime signals**—Apex/Flow invocation traces, error events, limit breaches, and field-usage analytics.
- **Trend and alert on entropy KPIs**—change-entropy, error rate, performance, limit usage.
- **Act in short, data-driven sprints**—prune dead Flows, consolidate objects, tighten governor limits.
- **Verify** improvements with the same telemetry that flagged the hotspot.

Complexity remains a badge of business success, but observability converts it from a lurking liability into a controllable variable. With a runtime lens always on, even the largest Salesforce org can evolve like a well-kept garden—dense, fruitful, and still navigable—rather than sliding into an untamed jungle of hidden vines and deadwood.

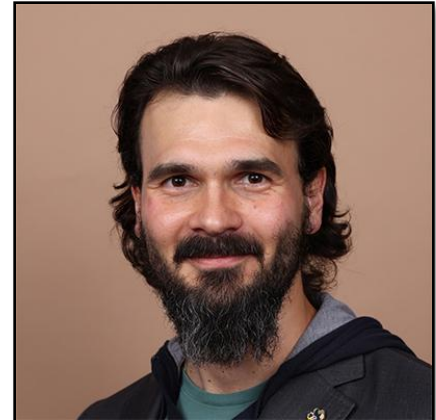


Your Org on Observability

Sources: This whitepaper references insights from software engineering research and industry analysis, including Lehman’s laws of software evolution [15], empirical studies of software entropy in codebases [16, 17], and Salesforce-specific guidance on org complexity growth [18]. Observability principles were discussed with reference to expert articles on managing microservices complexity [19] and data platform entropy [20]. These sources and others are cited in context above to support the conclusions drawn.

About the Author

Nikita Prokopenv is the founder and CEO of Pharos AI, the leading observability platform built natively on Salesforce. With more than two decades of hands-on engineering experience, Nikita is a recognized expert in Salesforce architecture, platform security, and application performance.



Nikita began his career at Salesforce, where he spent nearly nine years in R&D working on foundational technologies including Apex, Visualforce, My Domain, and the first cloud security scanner. His early contributions helped shape the metadata API and the secure, extensible platform Salesforce is today.

After leaving Salesforce, Nikita led complex implementation projects as a consulting architect before launching Pharos AI to tackle one of the platform's most persistent pain points: lack of visibility into what's really going on inside your org. Today, Pharos is helping teams intercept errors before they impact users, debug complex automations, and deploy with confidence.

Nikita combines deep technical expertise with a passion for developer productivity and org resilience. His work is driven by a belief that better tools make better teams and that observability is the next frontier for serious Salesforce professionals.

Follow Nikita on [LinkedIn](#)

About Pharos AI

Pharos AI is a Salesforce-native observability platform built by former Salesforce developers and engineers. It empowers technical teams with contextual logs, real-time monitoring, and intelligent alerts—right inside Salesforce.

To learn more, visit pharos.ai or reach out at info@pharos.ai.

Follow Pharos AI on [LinkedIn](#)