

SHAWN DAVISON, CTO, DEVIQ

MODERNIZING SAAS

From Legacy Apps to AI-Driven
Platforms



Modernizing SaaS: From Legacy Apps to AI-Driven Platforms

Author: Shawn R. Davison, CTO, DevIQ



Introduction

After a decade or more of growth, many midmarket SaaS companies (typically 10+ years old and \$100M+ in revenue) in industries like InsurTech, HealthTech, and MfgTech now find themselves at a turning point. They have achieved success over the years, yet now grapple with legacy technology, evolving customer expectations, and software engineers with outdated skill sets. Their **core challenges** include archaic architectures and accumulated technical debt that slow innovation, a tangle of fragmented systems from siloed acquisitions, and organizational silos that impede agility.

Meanwhile, investors and customers are pressuring these firms to infuse modern capabilities—especially Generative AI—into their products, lest they be left behind [1]. This eBook examines these challenges and outlines a strategic path forward. We believe that partnering with seasoned external experts and embracing a platform mindset can transform traditional SaaS products into more scalable platforms that generate more revenue, and enable ai-augmentation assuming it makes sense for the business.

Challenges Facing Mature SaaS Firms

- **Legacy Architectures & Technical Debt:** Many established SaaS companies are built on aging monolithic systems or outdated tech stacks. Over time, constant patching and feature additions have made these systems *fragile, costly, and difficult to enhance*. In fact, industry surveys find that the *complexity of legacy systems is often the top organizational challenge* in modernization initiatives [2]. These monolithic or tightly-coupled systems hinder rapid feature delivery and integration of new technologies. Teams spend disproportionate effort maintaining old code, with estimates that **30–50% of developer time** goes to legacy maintenance [3], or worse—productive maintenance time continues to plummet, especially with resource churn, and development slows to a crawl. Such technical debt diverts resources from innovation—a heavy price as more agile competitors enter the market.
- **Organizational Silos & Culture:** In many cases, the technology silos are mirrored by organizational silos. Teams accustomed to a specific product or tech stack may resist change. The company’s org structure (often a collection of independent product units from each acquisition, or rigid functional departments) may not encourage cross-functional collaboration. Conway’s Law tells us that systems mirror the communication structures of organizations – a siloed org will naturally produce siloed software. This can lead to finger pointing when there is an issue, leading to more delays, confusion, and lacking clear remediation paths. Thus, even if leadership envisions a modernized, integrated platform, internal culture and team structure can be a barrier. Product and engineering staff who “have not kept up with the latest development tools” may lack the skills or mindset to adopt new tech quickly. There may also be *talent gaps*, where legacy-tech experts abound but modern cloud/AI experts are in short supply – leading to resistance or fear of the unknown.



Figure 1 - Impediments to SaaS Growth and Efficiency

- **Overly-Complex Microservices (The “Distributed Monolith”):** In response to legacy pain points, some SaaS firms attempted to break apart monoliths into microservices. Microservices promise independent deployability and scalability for each service. However, without proper governance, teams can end up with a “*distributed monolith*” – microservices that are so interdependent they function like a tangled

single unit [4]. In other words, “*microservices can create technical debt, leading to distributed monoliths*” if not managed carefully [5]. Many organizations find it **nearly impossible to completely eliminate the old monolith**, carrying remnants that each microservice still relies on [6]. The result is an overly complex architecture: dozens of services, each with its own database and APIs, but still tightly coupled. This complexity can **slow development to a crawl**, as every change risks cascading effects. It also increases operational burdens in monitoring, testing, and coordinating deployments. In short, these firms are stuck with the liabilities of incomplete transitions – suffering both monolithic and microservice pains, instead of enjoying the benefits of either.

- Siloed Systems from Acquisitions:** Success over a decade often means growth through acquisitions. In insurance or healthcare tech, a company might have acquired smaller startups or legacy vendors to expand its offerings. But many companies never fully integrate these acquisitions technically. The outcome is *siloed product lines*, each with its own database, codebase, and sometimes duplicate functionality. Post-merger IT integration is notoriously hard – organizations “inherit multiple IT systems” and end up “*laboring under the siloed management*” of different processes and data [7]. For a SaaS provider, this could mean customer data and features live in separate silos by product, preventing a unified customer experience. Data trapped in silos cannot easily be shared or leveraged for insights across the suite. Organizationally, acquired teams may also remain semi-autonomous, reinforcing a siloed culture. These fractures make it difficult to execute company-wide improvements or cohesive platform strategies.

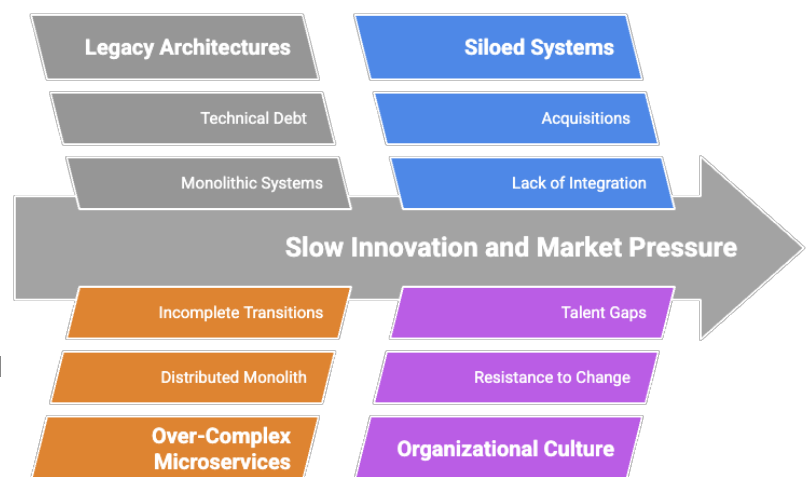


Figure 2 - Common Challenges

- Market Pressure to Adopt AI:** On top of internal challenges, these SaaS firms face external pressure to innovate, particularly with AI. The rapid rise of Generative AI in the last two years has created a sense of urgency. Investors and boards are asking: “What’s our AI strategy?” Executives feel the need to show they are infusing AI into the product roadmap. According to industry analysts, CIOs and CEOs are “facing pressure to adopt AI technology and articulate AI strategies” for their software products [8]. Over 65% of businesses now report using generative AI regularly – nearly double the share from a year prior – indicating how quickly AI has moved from novelty to mainstream. This hype cycle creates anxiety for mid-sized SaaS

providers: if they don't add intelligent AI features, their solutions might appear stale and get displaced by more "intelligent" competitors. Yet adopting AI is non-trivial, and doing it poorly (for example, bolting on a generic chatbot without proper integration) could backfire. Thus, management feels stuck between the *demand to innovate fast* and the *reality of slow, brittle systems*.

The Case for External Partnership



Given the above challenges, these companies often realize they **“don’t know what they don’t know”** about modern tech. Their engineering teams are talented and deeply knowledgeable about the product and industry, but may lack experience with the latest cloud architectures and AI developments. This is where **outside partners** can play a pivotal role. Bringing in external experts – whether specialized consultants, system integrators, or architectural advisors – injects fresh perspectives and up-to-date skills. These partners have seen patterns across industries and can cross-pollinate best practices.

Crucially, partners can accelerate change in ways that might be hard to achieve purely internally. They can audit the tech stack objectively, identifying quick wins to reduce tech debt. They can mentor internal teams on modern tools and methods (essentially acting as an “Enabling team” in Team Topologies terms, which we’ll discuss later). The need for outside help is underscored by industry data: **66% of organizations plan to involve external partners in their application modernization journeys**, either to guide internal teams or to execute the modernization directly [9]. In other words, two out of three companies modernizing their software are not going it alone – they recognize that external expertise is often the key to success.

Why are outside partners so important? For one, they have **deep technical and architectural experience** spanning multiple domains. A SaaS firm in HealthTech, for example, could benefit from a partner who has modernized systems in finance, telecom, and retail – gaining cross-industry insight. These partners also bring *hands-on experience with the latest frameworks and tools* – whether containerization and serverless, DevSecOps practices, or AI integration – which internal teams may not have had time to fully explore.

Another benefit is that external experts can act as **change agents**. Internal politics or inertia often slow down transformation; an outside team can cut through some of that by providing unbiased recommendations and steering the process with proven frameworks. This can help break confirmation bias to identify inefficiencies in process, highlight optimization opportunities, and eliminate waste. They can also shoulder part of the development load, allowing the internal team to keep focus on current product needs (so the business doesn't come to a halt during the modernization). It's akin to having a "tiger team" focused on the future while internal devs keep the lights on.

Importantly, partnership doesn't mean outsourcing everything. The most effective approach is collaborative: partners working *with* internal teams. This ensures knowledge transfer and buy-in, so internal engineers upskill and can maintain the modernized systems long-term [10]. In summary, **engaging external partners** with strong technical and cross-industry expertise can significantly de-risk a major modernization program. These partners provide the architectural vision, hands-on skills, and change management experience needed to break out of stagnation. For a SaaS firm striving to evolve, the takeaway is that **you don't have to do it alone** – and in fact, you probably shouldn't.

From Product to Platform: The Open API & GenAI Opportunity

Amid these challenges lies a significant opportunity: transforming a traditional SaaS product into a **platform**. Today's market winners are often platforms – systems that cultivate an ecosystem of users, developers, and partners, rather than just delivering isolated features. For a SaaS company, adopting *platform thinking* means designing your software to be extended, integrated, and built upon by others. This enables the business to scale, and opens up new revenue generating opportunities. Two key enablers of this shift are **open APIs** and **Generative AI**.

Open APIs as a Platform Driver: Opening up robust APIs (Application Programming Interfaces) allows third-party developers, partners, or customers to integrate your service with other products and even build new capabilities on top of it. This can exponentially increase the value of your SaaS. *Many successful SaaS companies have discovered that their API can generate revenue streams far beyond their core product. By creating extensible, monetizable APIs, you're not just selling a service — you're building a platform that can scale exponentially [8].* In other words, an API turns your product into a **hub** in a larger network of applications. For example, an InsurTech SaaS with open APIs might let

brokers integrate its policy system into their own portals, or let analytics firms plug in and crunch insurance data, spawning a whole ecosystem of connected solutions.

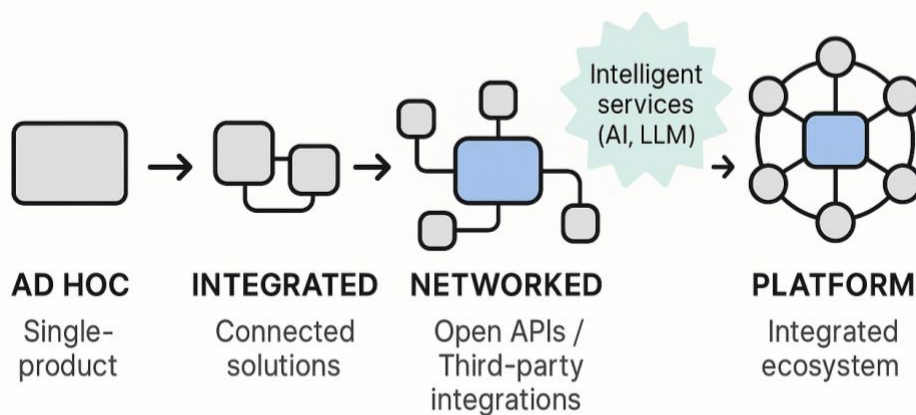
Open APIs also foster innovation by external developers. Rather than you alone coming up with every new feature, you allow others to extend your platform in creative ways. Think of Salesforce: it started as a CRM product, but by opening up APIs and a developer marketplace (Force.com), it became a *platform* with a vast array of third-party extensions. Midmarket SaaS firms can follow this path on a smaller scale. Even without a full marketplace on day one, having open APIs means your product's data and functions can interoperate with the wider digital world. This is crucial in industries like healthcare and IoT, where interoperability is a selling point. From a sales perspective, APIs also make your solution **stickier** – once a customer has integrated your API deeply into their workflows, it's much harder for them to rip-and-replace your system with a competitor's.

Generative AI as a Differentiator: Generative AI (GenAI), leveraging large language models (LLMs), offers new ways to enhance SaaS offerings and even create platform-like dynamics. Firstly, GenAI can be embedded as *features for end-users*. For instance, a HealthTech SaaS might integrate an LLM-based assistant that helps doctors auto-generate clinical notes or suggests documentation for treatment plans based on patient data. An IIoT platform could include an AI agent that analyzes sensor readings and natural-language maintenance logs to generate predictive maintenance plans. These kinds of AI-driven features can greatly increase the value users get from the platform, and they improve with more data – thereby encouraging users to centralize their work on your platform.



Secondly, GenAI can expose new **APIs or developer hooks**. For example, some SaaS providers now offer API endpoints for AI-driven services (often called “tools” or “schemas,” such as an API where you feed in a query and get a GPT-based answer drawing on the customer’s data in the platform). This again fosters an ecosystem: other products might call your AI APIs to add intelligence to their own offerings. We’re seeing the early stages of “*AI as a platform*” where SaaS companies provide AI capabilities as a service to others. In the near future, having a **GenAI strategy** will be akin to having an API strategy – it’s a new surface for extension. Indeed, it’s predicted that over *70% of enterprise software solutions will harness AI-driven capabilities* in some form this year [9].

PLATFORM MATURITY MODEL



Many SaaS companies evolve from siloed solutions toward becoming a platform

Figure 3 - Platform Maturity Model Evolution

In summary, **transforming into a platform** through APIs and AI can unlock new growth for aging SaaS products. It turns a potential weakness (being older and feature-rich but somewhat stagnant) into a strength: you have a mature product that can serve as the foundation of a broader platform. The open API strategy unlocks ecosystem value, and Generative AI capabilities differentiate the platform with smart features and services. This combination can revitalize a firm’s market position, making it central to a network of users and partners rather than just a lone product. Of course, to get there requires the technical modernization and cultural shifts which we discuss next.

Evolving the Architecture: Monoliths, Microservices, and Beyond

To support a platform-and-AI future, the underlying software architecture must evolve. Let’s explore how architecture paradigms have shifted and where many SaaS firms stand today – often stuck in the middle of an unfinished evolution.

Monolith to Microservices (and the Pitfalls): A decade ago, the prevalent architecture was the *monolith*: one large codebase and database handling all aspects of the application. Monoliths are straightforward in the early days but become unwieldy as the system grows. In the early-to-mid 2010s, microservices emerged as the remedy. Companies like Netflix and Amazon popularized breaking applications into many small, independently deployable services aligned with business capabilities. Many companies attempted this transition in the mid-2010s, after Martin Fowler’s seminal 2014 article on microservices [24].

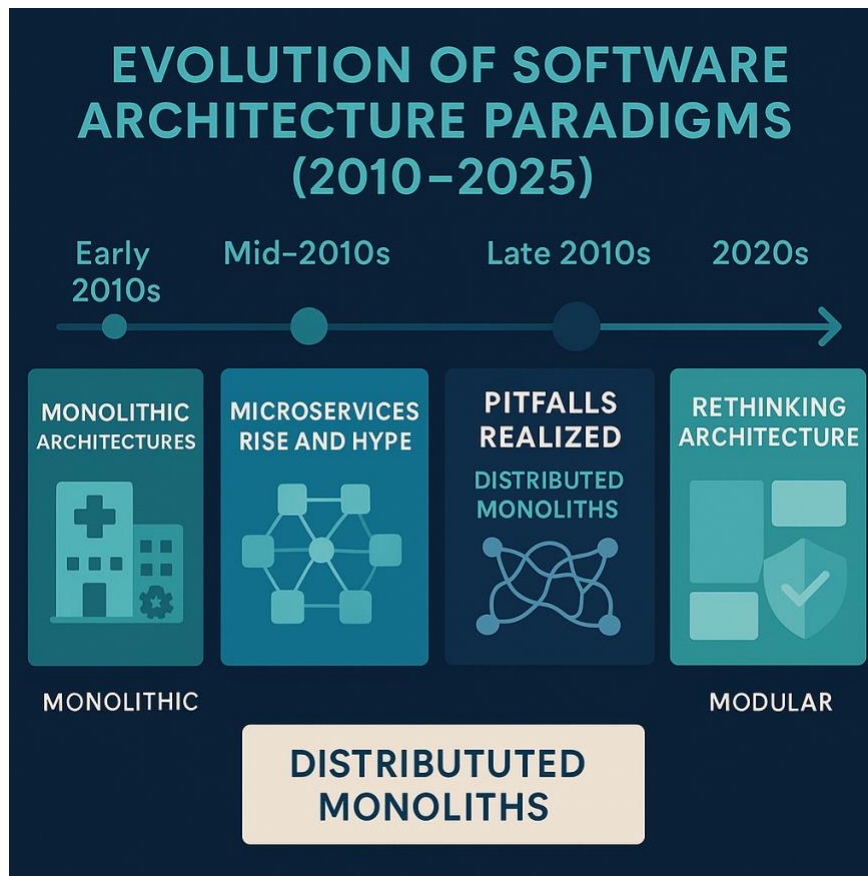


Figure 4 - Timeline of Microservices Adoption

Microservices, when done right, solve some monolith issues: teams can develop and deploy different components in parallel, scaling is more granular, and the system becomes more resilient to single-component failures [14]. However, *in practice many organizations struggled with microservices adoption*. Without careful domain design, they ended up with distributed complexity. An infamous scenario is the **distributed monolith**, where services are separated in name but not in reality – tightly coupled by synchronous calls or shared databases, such that a failure in one service can still cascade, and deployments require coordination. Many companies essentially “*replicated existing monolith issues in distributed form*,” gaining complexity without true independence [15].

Moreover, running dozens of microservices demands advanced DevOps maturity: automated deployments, container orchestration (e.g. Kubernetes), robust monitoring, and a culture of end-to-end ownership (“you build it, you run it”). Not every organization was ready for this. Thus we see incomplete transitions – perhaps a few services spun out of the monolith (e.g., an authentication service or a separate reporting module), but the core remains monolithic. Or a service was created for each major function, but they still share one database schema, negating many benefits. As one report noted, *“it is almost impossible to get rid of the monolith completely”* in such efforts [16]. The liabilities of this state are significant: higher latency between services, complex debugging across boundaries, increased operational overhead – all **without** achieving the agility that was the goal. In many cases, the result is a heavy ball-and-chain throttling feature velocity. The impact to the business—it can lead to a MORE risk-averse culture, regressing organizational maturity, agility, and even market positioning.

The Modular Monolith Alternative: For many SaaS firms, a more pragmatic approach than either a pure monolith or full microservices is the **Modular Monolith** pattern. This architecture maintains a single deployable unit (like a traditional monolith) but enforces strong internal modularity through clear boundaries and interfaces between components. Each module is designed as if it were a service – with its own domain model, data access layer, and well-defined API – but they all run in the same process. This provides many of the benefits of microservices (separation of concerns, team autonomy, clear ownership boundaries) without the operational complexity of distributed systems.

The key advantage of a Modular Monolith is that it avoids the pitfalls of both extremes. Unlike a distributed monolith, there’s no network latency between modules, no complex service discovery, and no need to coordinate deployments across multiple services. Unlike a traditional monolith, the codebase remains maintainable because modules are isolated and can be worked on relatively independently. Teams can own specific modules without stepping on each other’s toes, and the system can evolve toward true microservices later if needed (using the Strangler Pattern, discussed later).

A Modular Monolith is particularly well-suited for midmarket SaaS firms when:

- The system’s complexity doesn’t yet justify the operational overhead of microservices
- The team size is moderate (e.g., 5–20 developers) and can work effectively within a shared codebase
- The business domains are relatively stable and well-understood
- The organization wants to maintain deployment simplicity while still enabling team autonomy

The transition to a Modular Monolith can be more achievable than a full microservices migration. It often involves:

1. Identifying clear domain boundaries within the existing monolith

2. Refactoring the code to enforce those boundaries (e.g., via interfaces or dependency injection)
3. Establishing module ownership and governance practices
4. Implementing proper testing and CI/CD pipelines for the modularized structure

This approach can serve as a stepping stone – either as a permanent architecture for simpler systems, or as an intermediate state while gradually strangling out true microservices for the most independent components. The key is recognizing that not every system needs to be fully distributed to achieve maintainability and team autonomy.

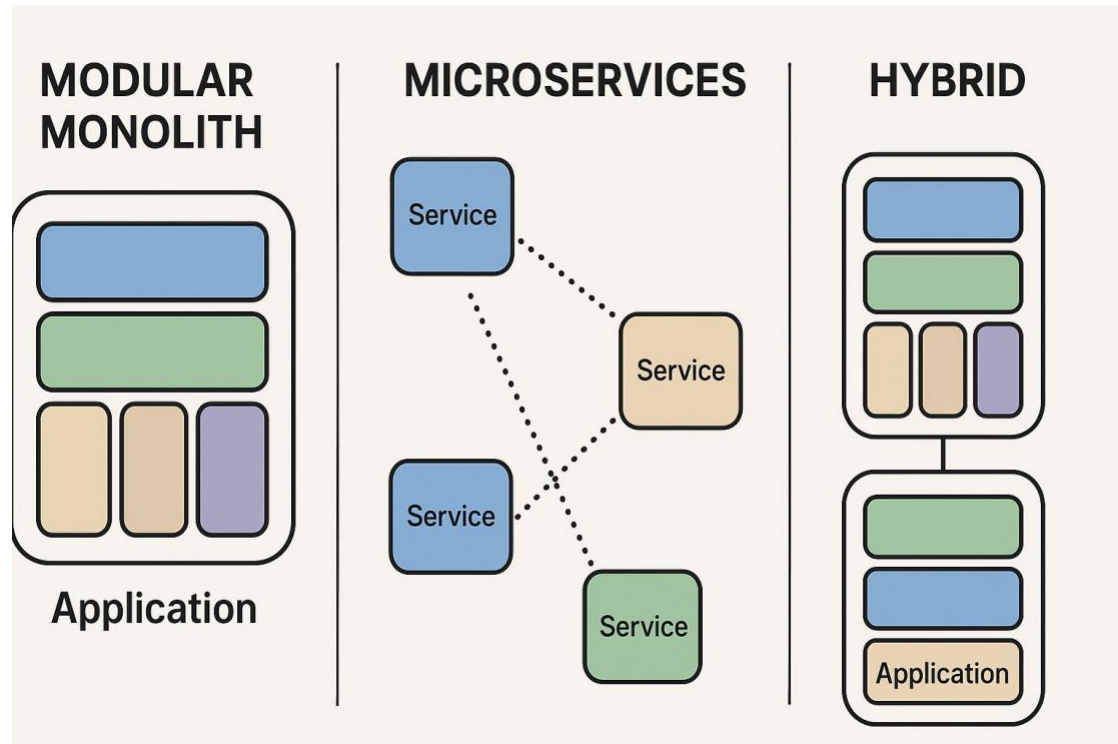


Figure 5 - Modular Monolith vs. Microservices

Does this mean microservices were a mistake? Not necessarily – but they need the right scope and execution. Some organizations have even begun **reversing course** (a recent trend of “monoliths are back” has been noted [17]). In reality, the answer is to find the *right-sized modularity*. For many companies, that might mean consolidating some services that were split too finely, while continuing to break truly independent domains out of the monolith. Modern practices like **domain-driven design (DDD)** and **bounded contexts** help in identifying proper service boundaries that align with business capabilities, so you avoid spaghetti calls between services. In any case, recognizing and refactoring a distributed monolith is a priority to reduce needless complexity.

Data Architecture: From ETL to ELT to Data Mesh: Alongside application architecture, data architecture has evolved. Many older systems follow a classic **ETL** (Extract,

Transform, Load) approach for analytics: data from the production database is extracted, transformed (cleaned and aggregated) on a schedule, and loaded into a data warehouse or data mart. This often results in overnight batch jobs, high latency in data availability, and rigid schemas. Modern data platforms favor **ELT** in cloud data warehouses or lakes: extract and load the raw data first (often via near real-time streaming), then transform on-demand as needed (the “data lakehouse” concept). This approach offers more flexibility and faster insights. However, adopting ELT requires newer tooling (e.g. change data capture, and cloud-based data lakehouse platforms) that some firms haven’t fully implemented. They may still be running legacy ETL pipelines on an on-premises database, limiting their ability to leverage data quickly for AI and analytics.

Going further, the latest trend is **Data Mesh** – an architectural paradigm for large-scale data management. Data Mesh, as coined by Zhamak Dehghani, “is a sociotechnical approach to sharing, accessing, and managing analytical data in a decentralized fashion.” [30] It applies microservice-like thinking to analytic data: instead of one central data team or warehouse, you have **domain-oriented data products** owned by cross-functional teams, and a self-serve data infrastructure platform that provides tooling for discovery, quality, and governance [18] [19]. The idea is to overcome the bottlenecks of a centralized data team and make data more usable across an organization. For example, an InsurTech might have separate data products for “Policy Sales Data,” “Claims Data,” and “Customer Profile Data,” each owned by the respective domain teams but interoperable through standardized interfaces – rather than all data being dumped into one giant warehouse managed by a single BI team.

Most companies are far from a true Data Mesh (this concept is still emerging, even in tech-forward enterprises). But many are stuck in partial transitions in data as well – maybe they moved some reporting to the cloud but still have siloed data stores per product, or they adopted a data lake but never curated it into a useful form (leading to a “data swamp”). The *liability of incomplete transition* here is that they cannot fully exploit the data they have. For instance, if acquisitions brought in separate databases, the company might never have unified them to get a 360° view of customers. That impedes advanced analytics and AI model training, which require consolidated, high-quality data.

DevOps and Infrastructure Modernization: The evolution also spans how software is built, tested, and deployed. Ten-year-old companies likely started with on-premises servers or basic IaaS hosting, manual deployments, and a clear separation between development and operations teams. Modern SaaS operates with Infrastructure as Code, CI/CD pipelines, containers, and cloud-native services. Many companies have begun adopting these (Docker, cloud CI servers, infrastructure provisioning scripts) but not uniformly. You often see *pockets of modernization*: for example, one new microservice is on Kubernetes, but the core monolith is still running on a manually configured VM; or automated tests exist for some modules but not others. Again, incomplete adoption can incur much of the cost (maintaining two deployment paradigms, retraining staff) with little benefit (still can’t deploy on demand because the old monolith on a VM is a bottleneck).

To truly benefit, modernization must be holistic: migrating infrastructure to a scalable cloud setup, implementing continuous integration and delivery so that *release frequency increases*, and ensuring observability (logging, monitoring, tracing) across both old and new components. Without these, adding fancy microservices or AI features on top of a brittle foundation will not succeed. It's telling that **95% of organizations say modernization is essential for success** [2], but execution must cover data, app architecture, and deployment processes together.

In short, **the current state for many SaaS firms is an architectural limbo**. They carry monolithic baggage *and* microservice complexity; they have batch-oriented data pipelines but aspire to real-time insights; and they operate with semi-automated DevOps processes that still have manual steps. This mixed state incurs extra overhead everywhere: duplicate systems, rework, synchronization issues, and skill mismatches. Recognizing this is the first step – the next is to systematically **complete the evolution** so the company can fully leverage modern paradigms (cloud, distributed systems, real-time data) without the drag of the past. In most cases, this is addressing an incomplete Microservices migration, and battling with a Distributed Monolith. It may be that existing resources don't know any different, because "this is the way it's always been", which is a clear tell that external expertise is needed to evolve. In the next section, we discuss how embracing Generative AI fits into this modernization journey, and how it can even help *reduce* technical debt when applied internally.

Embracing Generative AI for Velocity and Quality

Generative AI isn't just a product feature; it can also be a powerful accelerator for the *software development process itself*. For SaaS teams that need to increase delivery velocity and tackle technical debt, integrating AI into their engineering toolbox is increasingly a must. Let's explore a few avenues: LLM-powered coding assistants, AI-driven development tools, and AI in testing and operations.

LLM Coding Assistants (AI IDEs): The rise of tools like GitHub Copilot, Amazon CodeWhisperer, and others has shown that AI can help write code. These systems are essentially large language models trained on vast amounts of source code, integrated into the IDE to suggest code snippets or even entire functions as the developer types. The impact on developer productivity can be significant. For example, Microsoft reports that GitHub Copilot (powered by OpenAI's Codex model) is delivering "*40%+ improvements in developer productivity*" for coding tasks [20]. Developers can auto-complete boilerplate code, generate repetitive code constructs, and even get suggestions for complex algorithms in real time. For our aging SaaS firm's

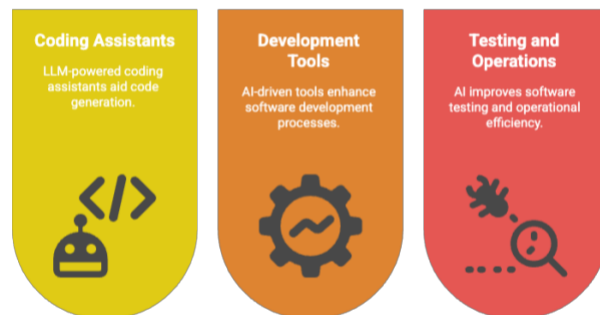


Figure 6 - AI Development & Testing Productivity

experienced developers, this is like having an ever-present pair programmer who knows all languages and frameworks. It can especially speed up work in unfamiliar tech – e.g. if the team is new to a particular cloud API, the AI assistant can suggest the correct usage patterns.

Faster coding means **faster feature delivery** – a critical metric these firms need to improve to satisfy customers and compete. Where a team might have taken weeks to add a new module, they might do it in days with AI assistance handling the rote parts. Moreover, it lowers the barrier to adopting new frameworks or tools.

AI-Assisted Code Refactoring: One reason technical debt persists is that refactoring legacy code is tedious and time-consuming, often with unclear short-term ROI. Generative AI can change that calculus. Researchers and companies are developing AI tools that analyze legacy code and propose refactor changes or even convert code from one language to another. For instance, AI can identify duplicate code, or identify inefficient code that is likely manifesting as performance issues or end user friction. According to analysis by AlixPartners, applying AI to code refactoring can yield a *1–3× reduction in the time* required, and about a 15–20% reduction in associated labor costs [21]. Essentially, what might have taken a team 3 months to painstakingly refactor by hand could potentially be done in a few weeks with AI-generated suggestions and automated fixes, supervised by engineers.

This has profound implications for reducing technical debt. It means those “back burner” cleanup projects might finally become feasible within a sprint or two. Imagine an AI tool scanning a sprawling monolithic codebase and highlighting 100 spots where modularization is needed – or even writing new microservice stubs to extract those pieces. While AI is not a silver bullet and its output needs human review, it can do a lot of heavy lifting (e.g. renaming variables consistently or migrating usage of an old library to a new one across thousands of lines). Some companies are already using AI to translate legacy code (like COBOL or outdated Java) into modern languages, preserving the logic while updating the syntax. This kind of automation can *eliminate chunks of tech debt* that would otherwise consume scarce senior dev hours.

Automated Testing and QA with AI: A huge part of maintaining velocity is ensuring quality through testing. AI is making inroads here as well. Smart test-generation tools can create unit tests or regression tests by analyzing existing code. Gen AI agents (e.g. Cursor) can simulate user interactions for UI testing or enumerate edge cases that developers might miss. Survey data shows that more than half of organizations are exploring or integrating AI into their test automation processes [22]. AI-augmented testing can *increase test coverage and speed up test creation significantly*. One source notes that AI in test automation helps generate tests **twice as fast** and boosts QA efficiency to keep up with rapid release cycles [23].

For our context, this means a lean team can achieve a level of testing rigor that previously only large organizations enjoyed. They can catch more bugs early, reducing the costly cycle

of bug-fix-release-repeat. It also means when they do refactor code or introduce new APIs, AI can quickly produce regression tests to ensure nothing breaks – addressing the common fear that “touching legacy code will introduce bugs.” Overall, AI in QA can improve reliability and confidence, which in turn lets the team ship faster.

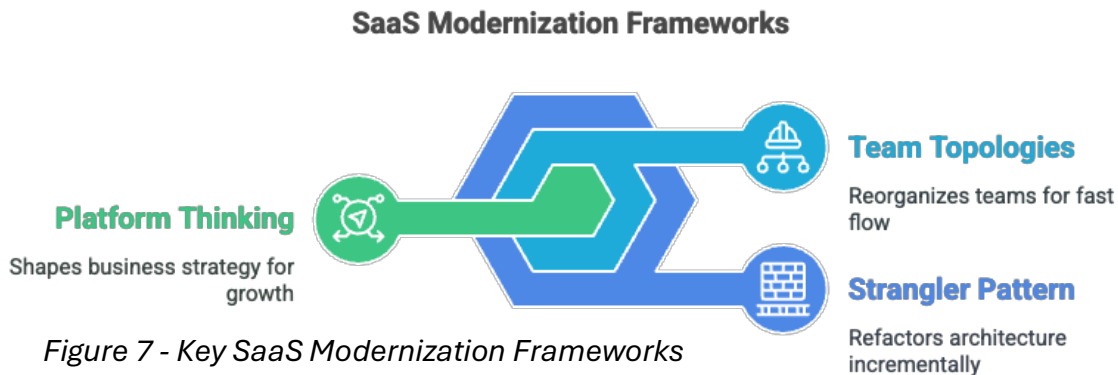
AI for DevOps and Monitoring: Beyond writing and testing code, AI can help in operations. Modern AIOps platforms use AI/ML to detect anomalies in system logs or metrics, predict outages, and even auto-remediate certain issues. SaaS Ops teams can leverage these to manage complex microservices environments more effectively – for example, an AI system might analyze log patterns and alert the team that a certain microservice is likely memory-leaking (something a human might not catch until a crash happens). AI can also optimize cloud costs by analyzing usage patterns and suggesting rightsizing of instances. All these contributions lead to lower downtime and more efficient operations, which again frees up time to focus on product improvements.

In sum, **Generative AI and related tools can be a force-multiplier for development speed and code quality.** By leveraging AI coding assistants, companies can drastically cut development and refactoring time, thereby increasing feature velocity – a key promise to impatient investors and customers. At the same time, these tools help reduce technical debt by making tasks like refactoring and testing less onerous, tackling the legacy burdens that slow the team. It’s worth noting that the *cultural aspect* is important: developers need training and encouragement to effectively use these AI tools in their workflow. But given that many routine programming tasks can be automated or accelerated, ignoring this trend would be a missed opportunity. Companies should pilot these AI tools in their development process as part of the modernization journey. Those that do are seeing tangible results (like the 40% productivity boost with Copilot), which can be the difference between meeting a quarterly release goal or not.

Finally, using AI internally creates a positive feedback loop for using AI in the product: the more comfortable the team is with AI, the more creatively they can think of applying it for their customers. It builds an AI-forward culture within the company – exactly what leadership and investors want to see.

Modernization Frameworks for Systematic Change

Tackling such a multi-faceted transformation – architecture, data, team structure, and AI enablement – requires a methodical approach. Three frameworks can guide midmarket SaaS firms on **how** to modernize systematically: the **Strangler Pattern** for incremental architectural refactoring, **Team Topologies** for reorganizing teams toward fast flow, and **Platform Thinking** (discussed earlier) for shaping the business strategy. Let’s delve into how applying these can turn a high-level modernization vision into an executable plan.



Incremental Refactoring with the Strangler Pattern

A common mistake in modernization is attempting a “big bang” rewrite – starting a new system from scratch to replace the old. This approach often fails due to unforeseen complexity and the difficulty of replicating years of business logic quickly. The **Strangler Fig Pattern**, introduced by Martin Fowler, offers a safer path: gradually build a new system around the edges of the old one, and piece by piece *strangle* (replace) the old system’s functionality [11]. The name comes from the strangler fig vine that grows around a tree and eventually replaces it.

The **key advantage** of this pattern is risk mitigation: *“The most important reason to consider a strangler fig [over a rewrite] is reduced risk. A strangler fig can give value steadily and frequent releases allow you to monitor progress more carefully.”* [11]. You can deliver new features (on the new platform) incrementally, and if something goes wrong, you only roll back that piece, not the entire system. It also means you can prioritize high-impact areas to modernize first, delivering business value early (which helps maintain stakeholder buy-in). Fowler notes that even if you stop halfway, you at least have those new components in place providing value, *“which is more than many cut-over rewrites achieve.”* [11].

For our midmarket SaaS firm, a strangler pattern implementation could look like this: identify a few candidate components or services (e.g., in an IIoT platform, perhaps the device management module; or in an InsurTech product, the quote calculation engine) that would benefit from reimplementing using modern tech and have relatively clear boundaries. Over a series of sprints, build those out as separate microservices or new modules, and use a strangler proxy to start diverting calls to the new components. In this way, over, say, a year, the company could migrate core pieces to a new architecture with continuous integration – instead of a risky multi-year “big bang” rewrite done in stealth.

It’s worth noting that to do this effectively, one must invest in *integration and testing*. The new and old parts need to coexist during the transition. Techniques like contract testing (ensuring new services meet the same interface contracts) and robust regression testing

are vital – this is where the earlier discussion on AI testing tools could help. Also, the strangler pattern can be applied not just at the application layer but also at the data layer (one can gradually migrate data from old databases to new ones or data lakes, syncing in real-time until cutover) and even at the UI layer (incrementally replacing parts of a legacy UI with new micro-frontend components). It's a versatile mental model – essentially, **replace the airplane's engines one at a time in mid-flight, without the plane crashing**. That's the goal.

Team Topologies: Aligning Teams for Fast Flow

Technology changes alone won't succeed without the right team structure. **Team Topologies**, a framework by Matthew Skelton and Manuel Pais, provides a blueprint for reorganizing engineering teams to improve delivery velocity and reduce silos. The essence of Team Topologies is recognizing that *different types of teams have different missions* and deliberately structuring them that way can streamline development. The framework defines **four fundamental team types**: Stream-aligned, Platform, Enabling, and Complicated-Subsystem teams, along with three modes of interaction between teams (collaboration, X-as-a-Service, and facilitating) [13].



A **Stream-aligned team** is focused on a specific product, feature set, or user flow – essentially aligned to a value stream. It's an empowered, cross-functional team that can deliver end-to-end value to customers. Think of it as a mini-startup team within the company, responsible for everything from development to operation of their slice of the system. *“Stream-aligned teams focus on a single, impactful stream of work (a product or service or set of features)... empowered to deliver customer value as quickly, safely, and independently as possible.”* [13]. In many legacy orgs, teams were historically organized by function (separate dev, QA, and ops groups) or by technical layer. Stream-alignment instead organizes by **business capability** and gives that team full ownership, which greatly speeds up delivery (no more throwing work over the wall to another team).

A **Platform team**, in this model, exists to support the stream-aligned teams. Platform teams build and maintain internal services, tools, and infrastructure that *enable the stream teams to deliver with autonomy*. For example, a Platform team might offer a CI/CD pipeline as a service, a common logging or monitoring service, or a set of internal API components that all product teams use. Atlassian's summary notes: *“Platform teams enable stream-aligned teams to deliver work with substantial autonomy... providing internal services that the stream-aligned team can use.”* [13]. The goal is to reduce the cognitive load on stream teams – they shouldn't all have to be experts in cloud networking or Kubernetes internals if a platform team can provide a simplified platform for deployment and operations.

Enabling teams are like internal consultants or coaches. Their job is to help stream-aligned teams acquire new capabilities or overcome obstacles, especially in areas where expertise is limited. For instance, an enabling team might be a pair of UX design experts who rotate among product teams to uplift design practices, or a group of data scientists who help various teams implement AI features. They don't own a permanent part of the product, but they accelerate others. Skelton and Pais emphasize that enabling teams **increase the autonomy** of stream teams by growing their skills and knowledge, rather than creating long-term dependencies [14].

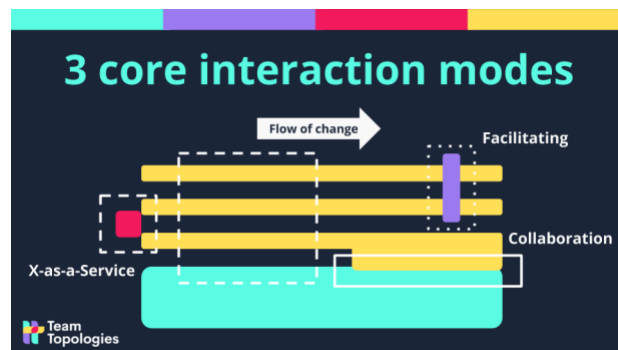


Figure 8 - Team Topologies - Interaction modes

The fourth type, a **Complicated-Subsystem team**, is a special case – a team that owns a particularly complex component that requires deep expertise (for example, an algorithm-heavy engine or a compliance-critical module). Instead of trying to spread that knowledge or embed specialists everywhere, you centralize that work in one team that others can interface with. Not every organization will have this need, but if, say, our InsurTech platform has a very complex actuarial calculation engine, they might isolate a team specifically for it.

By adopting *Team Topologies*, a SaaS company can break down organizational silos. Instead of teams formed by historical accident (e.g., one team per acquired product, each with its own full stack), you reorganize around the target-state product streams. Perhaps you want to offer one unified platform – you could create stream-aligned teams for major domains (for example, in HealthTech: an Appointments & Scheduling team, a Medical Records team, a Billing team, etc., cutting across what used to be separate products). Platform teams then support all of them (e.g., a Cloud Platform team, a Data Platform team). Enabling teams tackle cross-cutting concerns (like security, which is often siloed as its own department – instead, a security enabling team can embed with others to improve practices).

This approach also directly addresses the “**experienced teams vs modern tools**” gap: by forming enabling teams or hiring talent into platform teams, you inject new tool expertise and those experts help upskill the rest. It creates a learning organization.

From a *DevOps perspective*, Team Topologies is about Conway’s Law in action: design the organization structure to encourage the desired software architecture (fast-flow, loosely coupled services, etc.). Stream-aligned teams should correspond to microservices or modules that they own fully. Platform teams correspond to internal platform services. This alignment reduces hand-offs and wait times. In contrast, if you left a siloed “Database Team” separate from an “Application Team,” every schema change becomes a negotiation across silos – slow and frustrating. The Team Topologies approach would instead have a

cross-functional team that includes database and application skills working together on one stream.

In short, **restructuring the teams may be as critical as rewriting the code**. As one expert succinctly put it, organizing business and technology teams for “*fast flow of value*” is the core of Team Topologies (see *Team Topologies – Organizing for fast flow of value*, by Skelton & Pais). Firms should assess their org chart: Do we have these four types of teams in the right places? Are our team interactions healthy (e.g., platform teams offering services rather than acting as gatekeepers)? Adopting this model can accelerate modernization because the *people* side aligns with the *technology* side.

Platform Thinking and Continuous Evolution

We already discussed platform thinking from a business-model perspective (opening APIs, fostering ecosystems). Here, let’s tie it into the modernization journey itself: thinking of your system as a platform means **building it to continuously evolve** and support new extensions. It encourages **modularity**, because a platform exposes services that might be consumed in unforeseen ways. It encourages **open standards and APIs**, because you want external players to connect easily. And it emphasizes **developer experience**, since a successful platform needs great documentation and ease-of-use for those building on it.

Practically, adopting platform thinking could mean instituting a formal API-first design approach for all new features (designing and documenting APIs using standards like OpenAPI/Swagger before coding begins). It could mean creating sandbox environments or developer portals for partners by the time you launch new APIs. Culturally, it means treating internal components as if they were products too – e.g., your platform team provides a logging service and treats the developers in stream teams as their “customers” to satisfy. This inner-sourcing attitude (developing internal platforms with product discipline) has been shown to yield better internal tools and, consequently, faster delivery for product teams (as noted in Team Topologies case studies).

Platform thinking also complements the *Strangler Pattern* and *Team Topologies*. As you strangle out components, you design their interfaces cleanly as services/APIs that could be exposed more widely later. And your stream-aligned and platform teams map to providing those interfaces.

To ensure the modernization is systematic and doesn’t stall, these frameworks can work together: the Strangler pattern provides the *how* for technical migration, Team Topologies provides the *how* for people and process change, and Platform Thinking provides the *why* – the end-state vision of a more open, extensible, AI-powered SaaS platform that can continually adapt. Together, they help in planning a roadmap. For example:

- **Phase 1:** Identify two pilot domains to strangle out and form corresponding stream-aligned teams.
- **Phase 2:** Build out platform services for common needs.
- **Phase 3:** Open up APIs and engage early partners.

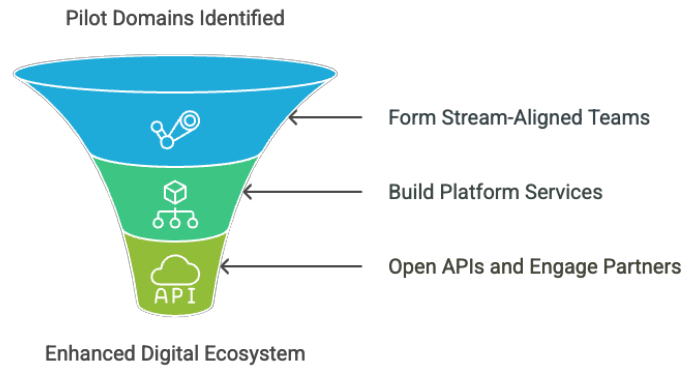


Figure 9 - Transformation Phases

Each phase delivers incremental value (new features, improved performance, etc.) while moving closer to the strategic vision.

One more concept worth mentioning is a **“strangler fig” approach to team structure**. While strangling the architecture, you can also gradually strangle (or evolve) the organization – shifting team responsibilities as new components come online. For example, initially a legacy Product Team A owns everything. Once you carve out a new microservice or module, you might create a new stream-aligned team to own that component (perhaps seeding it with members from Team A plus some new hires). Over time, Team A’s scope shrinks and the new team’s scope grows, until Team A might dissolve or be repurposed. This avoids the shock of a full reorg on day one; instead, org change happens fluidly alongside the tech change.

Integrating Generative AI into Legacy SaaS Platforms

As AI-native disruptors enter every software vertical, aging SaaS providers face a narrow window to modernize and defend their market. The integration of Generative AI (GenAI) – especially through open standards like the Model Context Protocol (MCP) – allows SaaS firms to rapidly embed intelligent capabilities without rewriting their core systems.

By exposing existing data and services through MCP, companies can plug in AI assistants, copilots, and agents that generate insights, automate workflows, and enhance UX. This enables faster innovation, improved customer retention, and new revenue streams – all while leveraging the incumbent’s data and domain expertise.

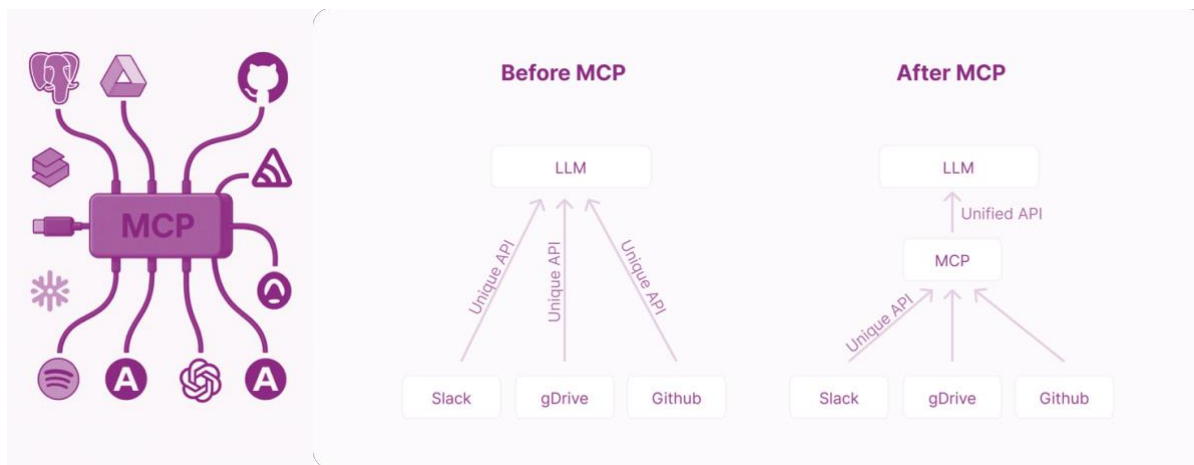


Figure 10 - MCP as the USB Hub for LLMs – a key enabler for SaaS AI integration

Model Context Protocol API Flow

Real-world examples like Slack GPT and Block’s use of MCP show how established platforms can layer GenAI features into legacy stacks to maintain leadership. Slack, for instance, integrated OpenAI and Anthropic LLMs via its existing app framework to deliver features like automatic conversation summaries and AI-powered writing assistance. Similarly, Block (formerly Square) adopted MCP to securely connect AI agents with its transactional data and services, enabling intelligent financial assistants and automated anomaly detection.

But tech alone isn’t enough. This evolution requires Sociotechnical Leadership: aligning architecture, org structure, and culture with AI-driven goals. External partners with deep GenAI experience play a critical role in accelerating transformation, mitigating risk, and enabling lasting change. Many organizations are even creating new leadership roles like Chief AI Transformation Officer to oversee this strategic shift.

Bottom Line: GenAI and open standards offer a strategic leap for SaaS incumbents—but only if they act now. The window for incumbents is open but closing fast. By embracing approaches like MCP and LLM integration, companies not only delight customers but also send a message to the market that they will not stand still. Speed matters: those who experiment and iterate with GenAI features today will refine them into core differentiators tomorrow.

Industry Spotlights: Modernization in Action

To ground these ideas, let's consider how they apply in the specific verticals of **InsurTech**, **HealthTech**, and **Industrial IoT (IIoT)** – industries where our midmarket SaaS firms operate. Each has unique drivers and provides examples of modernization in practice:

InsurTech: Modernizing a Midmarket Leader

Insurance technology providers (InsurTechs) often face the challenge of serving conservative clients who themselves rely on massive legacy systems (e.g., mainframes, COBOL applications). Many midmarket InsurTech SaaS platforms, originally built 10–15 years ago, were early pioneers in cloud-based solutions such as enrollment management or digital quoting. However, the competitive landscape has evolved rapidly, with new startups now offering AI-driven underwriting, fraud detection, and customer engagement tools.

InsurTech firms today must contend with complex data integration requirements – connecting to insurers' legacy systems, third-party data providers (like MIB, medical records, or driving histories), and digital broker channels. Modernizing these platforms typically involves breaking up monolithic architectures into microservices for quoting, policy management, billing, and claims – enabling faster, more agile updates and integrations.

Open APIs are essential. Insurance workflows naturally involve many partners – brokers, agents, medical providers, employer portals – all requiring secure and efficient access to insurance services. Leading platforms expose APIs for quoting, claims submission, and enrollment, enabling insurance services to be embedded into broader ecosystems. Additionally, the insurance industry increasingly views rapid adoption of Generative AI as a competitive necessity, particularly for automating customer interactions, claims processing, and underwriting decisions.

Case Study: MGIS Platform Modernization

MGIS, a leading provider of insurance services for healthcare professionals, faced the challenge of modernizing a core platform originally built on TenFold technology. While the TenFold system served as a reliable backend for many years, new market demands required a modern, customer-centric experience that could engage brokers, healthcare professionals, and internal teams through intuitive web portals and real-time integrations.

DevIQ partnered with MGIS to deliver a modernization program built on the Microsoft Azure platform, with custom .NET applications surfacing new products and experiences while maintaining integration with the legacy TenFold data store. Key initiatives included:



- Enrollment Portal: A modern B2C web experience for individual healthcare professionals to enroll in MGIS insurance products easily and securely.
- Broker Portal: A B2B portal enabling brokers to manage client enrollments, view policies, and generate quotes.
- High Limits Module: A specialized product offering high-limit disability coverage, integrated into the broader enrollment and quoting workflows.
- MedTravel Insurance Module: A new offering allowing healthcare professionals to secure insurance coverage for travel-related risks, embedded within the digital experience.
- IPHL Quote Module: A streamlined quoting tool for Individual Professional High Limits (IPHL) products, surfacing complex underwriting decisions through a modern user interface.

By architecting each of these capabilities as independent but integrated applications on Azure, MGIS created a modular and scalable ecosystem without needing to immediately decommission the TenFold core. This allowed MGIS to:

- Rapidly launch new customer-facing products.
- Improve broker and client engagement through modern UX/UI design.
- Enable future AI-driven enhancements, such as automated underwriting or claims support, without rearchitecting from scratch.
- Lay the groundwork for eventual migration of core backend processes to cloud-native solutions.

Over time, MGIS has shifted its market position from being a legacy insurance software provider to a modern platform company that can quickly respond to customer needs, regulatory changes, and new product opportunities.

HealthTech: Integrating fragmented systems

Healthcare software – such as EHR (Electronic Health Records) systems, practice management, or patient engagement platforms – historically suffers from extremely **fragmented systems** and strict compliance constraints. A HealthTech SaaS company that started a decade ago might have initially offered a niche solution (e.g., a telemedicine scheduler) then expanded. They likely had to interface with hospital EMRs or clinic systems, leading to a lot of integration code (and possible technical debt) in that layer. Healthcare



data standards (HL7, FHIR) also evolved over time – older systems might use legacy HL7 interfaces, whereas modern ones use FHIR APIs.

Healthcare providers have been slow to upgrade their systems – as of 2021, **73% of healthcare providers still used outdated legacy systems** in some capacity [17]. This means a HealthTech SaaS must often bridge old and new tech. Modernization for them might include adopting FHIR APIs for interoperability, containerizing their solution for cloud deployments that meet strict data residency rules, and ensuring high security (HIPAA compliance and beyond).

Generative AI in healthcare has big promise: summarizing patient visits, extracting insights from medical literature, or triaging patient messages. We've already seen early deployments of AI scribes that listen to doctor-patient conversations and draft clinical notes. A HealthTech SaaS could integrate such an AI to reduce physicians' administrative burden. For instance, an AI summarizer built into an EHR could auto-generate a patient's visit note, which the doctor then simply reviews and approves – saving time. AI can also help analyze large patient datasets to identify care gaps or predict outcomes, which forward-thinking platforms might offer as analytics add-ons. (A key requirement here is ensuring patient health information is handled in a compliant manner when used with AI.)

Consider a HealthTech SaaS case: **MediConnect** started in 2012 as a cloud-based scheduling and appointment system for clinics. By 2025, it acquired a small EHR vendor and a telehealth startup. Now it has a suite of offerings but on disparate architectures. Their modernization strategy is to create a unified *Healthcare Cloud Platform*. They use the strangler pattern to rebuild the patient records module as a new service using a modern stack, ensuring it's FHIR-native for easy data exchange. They reorganize teams into stream-aligned units like the "Patient Profile Team," "Appointment & Visit Team," etc., cutting across the old product boundaries. They leverage AI by incorporating an LLM that can take a doctor's voice recording of a visit and draft a structured summary into the system (using their new unified data platform to fine-tune the model securely on medical data). They also open APIs so that other digital health apps (say, a mobile wellness app) can pull scheduling or medical data (with patient consent) from MediConnect. By doing so, MediConnect becomes more of an open healthcare platform that clinics and partner apps can all use, rather than a set of siloed tools. This addresses the needs of hospital CXOs who want integrated solutions and are increasingly open to cloud-based, AI-driven healthcare IT.

MfgTech/IloT (Industrial IoT): IloT SaaS platforms connect physical devices (sensors, machines, vehicles) to cloud services for monitoring, analytics, and control. A midmarket IloT provider might focus on a vertical – e.g., factory equipment monitoring, fleet management (trucks, shipping containers), or smart building systems. Ten years ago, they



likely dealt with mostly on-premises deployments (factories historically were cautious about the cloud) and devices sending data to a central server via protocols like MQTT. Today, IoT has embraced cloud and AI for advanced analytics (predictive maintenance, anomaly detection).

Challenges in this space include scaling to millions of device messages, ensuring real-time processing, and handling big data (large volumes of time-series data). A legacy IIoT SaaS might have a monolithic application handling device management, data storage, and analytics all together. Modernizing it could mean adopting a microservices and data-pipeline architecture: separate services for device connectivity, real-time stream processing (maybe using tools like Kafka or Azure IoT Hub), and analytics engines. Data mesh ideas could even apply if different product lines manage different kinds of data. Tech debt often appears as custom device protocols or one-off logic for each client's machines – refactoring to standard protocols (MQTT, HTTPS APIs) and a plug-in adapter architecture can help.

Generative AI in IIoT can assist in making sense of data and aiding human operators. For example, an AI assistant might let a technician query the system in natural language: *“Which sensors showed anomalies this week in Plant 5?”* and get a synthesized answer with recommended actions. Or AI might automatically generate incident reports when machines fail, pulling in data from logs. We also see AI being used in computer vision on factory floors or for drone inspections, which tie back into IoT platforms.

A notable example: **Samsara**, a leading IIoT platform for fleet and physical operations, is *“applying AI to the proprietary data it collects from sensors in over a million vehicles to prevent accidents”*, leveraging a huge volume of driving data to improve safety [29]. This shows how an IIoT company can integrate AI at the core of its value proposition. A smaller midmarket IIoT firm could similarly train models on its domain data (say, vibration readings from machines) to predict failures and offer that as a premium AI-driven feature.

Case in point: **FactoryPulse** (fictional) provides an IoT platform for manufacturing equipment monitoring. It's 12 years old, initially delivered as on-premises software connecting to PLCs and SCADA systems on factory floors. Now they're shifting to a cloud-based subscription model. They use the strangler pattern to develop a new cloud microservice for real-time sensor data ingestion and processing, gradually migrating clients from on-site servers to cloud endpoints. They containerize an edge data collector that can be deployed on a small gateway device at the factory, which sends data to the cloud service – this isolates site-specific legacy code.

For their core application, FactoryPulse adopts a **Modular Monolith** architecture. This approach makes sense because:

- Their core functionality (device management, data processing, analytics) is tightly coupled but can benefit from clear internal boundaries

- Their development team is ~20 people, making a full microservices architecture overkill
- They need to maintain high performance for real-time data processing
- They want to enable team autonomy while keeping deployment simple

The Modular Monolith structure allows them to:

1. Maintain a single deployable unit for the core platform
2. Define clear module boundaries (Device Management, Data Processing, Analytics, etc.)
3. Enable different teams to own specific modules
4. Keep inter-module communication fast and reliable (no network latency)
5. Gradually extract truly independent services later if needed

Team Topologies comes into play as well: they form a platform team focused on the data pipeline and IoT device management, while separate stream-aligned teams focus on different industry solutions (one for automotive plants, one for chemical plants, etc., since these sectors have different needs but share the same platform). They incorporate AI by partnering with an ML company to integrate a predictive maintenance model – the platform team exposes this via an API so each industry-focused team can use it. They also add a feature where a plant manager can chat with an AI assistant: “Show me any abnormal readings in the past 24h,” and it will analyze the sensor streams and highlight issues. By modernizing in this way, FactoryPulse not only sheds its old image of clunky on-prem software, but can market itself as an *AI-enhanced IIoT platform* that improves operational efficiency. This appeals to both the CTO (technology improvement) and the CEO (new business value) at their client companies.

Across these industries, some common themes emerge from the case studies: incremental tech upgrades, using AI to deliver new value, breaking down silos for a unified platform experience, and involving partners or ecosystems. They highlight that modernization is not just a tech refresh – it can drive **business transformation**: new services, new revenue models (API monetization, AI features as add-ons), and deeper customer engagement.

Conclusion

For North America's SaaS providers in InsurTech, HealthTech, IIoT and beyond, the mandate is clear: **adapt and modernize, or risk irrelevance**. The challenges are substantial – legacy systems laden with technical debt, fragmented architectures from years of growth, and pressure to catch the AI wave – but these challenges are surmountable with a strategic, comprehensive approach. This eBook has outlined that approach.

By embracing *systematic modernization* – leveraging incremental refactoring via the Strangler Pattern, reorganizing teams around value with Team Topologies, fostering a platform mentality, and integrating GenAI through open standards like MCP – these companies can gradually shed the weight of outdated technology while continuously delivering value. Modernization is not an overnight flip; it's a journey of steady improvements. Each step (be it carving out a microservice, automating a pipeline, opening an API, or integrating AI capabilities) builds momentum and capability. As legacy constraints fall away, the benefits compound: **faster time-to-market, lower operational risk, improved scalability, and the ability to innovate with AI and data**.

Crucially, modernization is as much about people and mindset as technology. The companies that succeed empower their teams, welcome outside expertise, and create a culture of continuous improvement. They treat technical debt reduction and refactoring not as a one-time project, but as an ongoing practice (much like regularly paying down financial debt). They align business goals with IT capabilities, ensuring leadership understands that adding flashy AI features means little if the foundation is cracking – so they invest in that foundation. At the same time, they don't modernize in a vacuum; they keep an eye on delivering incremental business value at each phase (for instance, releasing an AI feature early on that wows customers, even as backend work continues behind the scenes).

The role of **external partners** cannot be overstated for mid-sized firms with limited R&D bandwidth. Engaging the right partners accelerates the journey and brings in skills that would take too long to build internally. Two-thirds of organizations rely on external help in their modernization – this is not a sign of weakness, but of wisdom in leveraging the broader tech ecosystem. With specialists handling complex cloud migrations or AI model integration, the internal team can focus on what they know best: the business domain and customer needs.

Looking ahead, transforming a product-centric SaaS company into a platform-centric, AI-augmented solution provider positions it for resilience in the next decade. It opens new growth avenues – partner marketplaces, data monetization, AI-driven services, to name a few. It also builds a competitive moat: a modern platform is easier to improve continuously, whereas rivals stuck on legacy will fall further behind as technology cycles accelerate. As Bill Gates famously observed, *“We always overestimate the change that will occur in the next two years and underestimate the change that will occur in the next ten.”*

The rapid doubling of generative AI adoption in the last year is a reminder not to be complacent. Ten years from now, today's nascent tech (AI, edge computing, whatever comes next) will be commonplace, and software providers need to lay the groundwork now to harness them.

In conclusion, aging (10+ year-old) SaaS companies have a tremendous opportunity to reinvent themselves. By addressing core technical and organizational challenges head-on and embracing modern architectures and AI, they can evolve from aging single-product vendors into dynamic **platforms** that define their industries' future. The path requires vision, partnership, and disciplined execution – but the reward is a business that is not only more efficient and agile, but one that can delight customers with continuous innovation. Modernization is not a one-time project; **it's a continuous journey** of alignment with the ever-changing technology landscape. With the right strategy, tools, and team, even a twenty+ year-old SaaS can achieve a “second act” – emerging as a tech leader of the 2020s, not a legacy of the 2010s. The message to SaaS CXOs is optimistic: the problems you face **do** have solutions, and by acting now, you can turn your legacy into leverage and your tech debt into dividends, propelling your company into its next phase of growth.



Bibliography

- [1] Baron Funds, "AI Hype and the Death of Software," 2024. [Online]. Available: <https://www.baroncapitalgroup.com/article/insights-ai-hype-and-death-software>
- [2] Red Hat, "The state of application modernization," 2024. [Online]. Available: <https://www.redhat.com/en/resources/app-modernization-report>
- [3] AlixPartners, "Can AI solve the rising costs of technical debt?," 2024. [Online]. Available: <https://www.alixpartners.com/insights/102jar/can-ai-solve-the-rising-costs-of-technical-debt>
- [4] N. Nayeem, "Why Microservices Are a Double-Edged Sword and How to Make the Right Architectural Choice," Medium, 2024. [Online]. Available: <https://medium.com/@nomannayeem/why-microservices-are-a-double-edged-sword-and-how-to-make-the-right-architectural-choice>
- [5] P. Coder, "Microservices Are Technical Debt," DEV Community, 2024. [Online]. Available: <https://dev.to/parmcoder/microservices-are-technical-debt-eh2>
- [6] VentureBeat, "Why microservices might be finished as monoliths return with a vengeance," 2024. [Online]. Available: <https://venturebeat.com/data-infrastructure/why-microservices-might-be-finished-as-monoliths-return-with-a-vengeance>
- [7] Acquisition International, "Siloes and Separations: The IT Challenges of Mergers and Acquisitions," 2024. [Online]. Available: <https://www.acquisition-international.com/siloes-and-separations-the-it-challenges-of-mergers-and-acquisitions>
- [8] Zuplo, "API Monetization for SaaS: Transforming Your Software into a Revenue Powerhouse," 2025. [Online]. Available: <https://zuplo.com/blog/2025/02/27/api-monetization-for-saas>
- [9] Blocktunix, "Generative AI in SaaS Development | Benefits & Trends | Predict," Medium, 2024. [Online]. Available: <https://medium.com/@Blocktunix/generative-ai-in-saas-development-powering-next-generation-enterprise-applications-5ab7951fa71d>
- [10] Woolpert, "Thinking in Platforms: The Many Trains of Thought," 2024. [Online]. Available: <https://woolpert.com/media/blogs/thinking-in-platforms-the-many-trains-of-thought>
- [11] M. Fowler, "Original Strangler Fig Application," 2024. [Online]. Available: <https://martinfowler.com/bliki/OriginalStranglerFigApplication.html>
- [12] Microsoft, "Strangler Fig Pattern – Azure Architecture Center," 2024. [Online]. Available: <https://learn.microsoft.com/en-us/azure/architecture/patterns/strangler-fig>
- [13] Atlassian, "Team Topologies," 2024. [Online]. Available: <https://www.atlassian.com/devops/frameworks/team-topologies>
- [14] Team Topologies, "Key concepts and practices for applying a Team Topologies approach to team-of-teams org design," 2024. [Online]. Available: <https://teamtologies.com/key-concepts>
- [15] IBM, "Insurance Leaders Agree That rapid Adoption of Generative AI is necessary to compete," 2024. [Online]. Available: <https://newsroom.ibm.com/2024-10-16-new-ibm-study-insurance-leaders-agree-that-rapid-adoption-of-generative-ai-is-necessary-to-compete>
- [16] McKinsey & Company, "The potential of generative AI in insurance," 2024. [Online]. Available: <https://www.mckinsey.com/industries/financial-services/our-insights/insurance-blog/the-potential-of-gen-ai-in-insurance-six-traits-of-frontrunners>
- [17] Devico, "Navigating Healthcare's Tech Debt," 2024. [Online]. Available: <https://devico.io/blog/navigating-healthcares-tech-debt>

- [18] ThoughtWorks, “Data Mesh Principles and Logical Architecture,” 2020. [Online]. Available: <https://martinfowler.com/articles/data-monolith-to-mesh.html>
- [19] Testfort, “Future of Test Automation Tools: AI Use Cases and Beyond,” 2024. [Online]. Available: <https://testfort.com/blog/test-automation-tools-ai-use-cases>
- [20] Microsoft, “*GitHub Copilot X: The AI-powered developer experience*,” Microsoft Build 2023, May 2023. [Online]. Available: <https://github.com/features/copilot>
- [21] AlixPartners, “Can AI solve the rising costs of technical debt?,” 2024. [Online]. Available: <https://www.alixpartners.com/insights-impact/insights/can-ai-solve-rising-technical-debt-costs/>
- [22] TestGuild, “Top 8 Automation Testing Trends Shaping 2025,” 2024. [Online]. Available: <https://testguild.com/automation-testing-trends>
- [23] Testfort, “Future of Test Automation Tools: AI Use Cases and Beyond,” 2024. [Online]. Available: <https://testfort.com/blog/test-automation-tools-ai-use-cases>
- [24] M. Fowler, “Microservices,” 2014. [Online]. Available: <https://martinfowler.com/articles/microservices.html>
- [25] M. Fowler, “Original Strangler Fig Application,” 2024. [Online]. Available: <https://martinfowler.com/bliki/OriginalStranglerFigApplication.html>
- [26] IBM, “Insurance Leaders Agree That rapid Adoption of Generative AI is necessary to compete,” 2024. [Online]. Available: <https://newsroom.ibm.com/2024-10-16-new-ibm-study-insurance-leaders-agree-that-rapid-adoption-of-generative-ai-is-necessary-to-compete>
- [27] McKinsey & Company, “The potential of generative AI in insurance,” 2024. [Online]. Available: <https://www.mckinsey.com/industries/financial-services/our-insights/insurance-blog/the-potential-of-gen-ai-in-insurance-six-traits-of-frontrunners>
- [28] Devico, “Navigating Healthcare’s Tech Debt,” 2024. [Online]. Available: <https://devico.io/blog/navigating-healthcares-tech-debt>
- [29] Baron Funds, “AI Hype and the Death of Software,” 2024. [Online]. Available: <https://www.baroncapitalgroup.com/article/insights-ai-hype-and-death-software>
- [30] N. Ford and Z. Dehghani, “*Software Architecture: The Hard Parts*,” O’Reilly Media, 2022.