The Lean App Movement

Optimize for outcomes that matter — not pointless, repetitive drudgery

Vendia

Table of Contents

Lean Apps

Build software based on customer value, rather than technological ownership

Summary

- Traditional applications, overweight "DIY", and implementation scope creates a false sense of control and change isolation; by contrast, lean apps focus on shifting as much undifferentiated work (server management, data sharing) as possible to platform providers who can deliver it more consistently and cost effectively using cloud-native, multi-tenanted SaaS implementations.
- Lean apps decrease development time and delivery risk by reducing application surface area to four key elements: *data model*, *integrity constraints*, *stateful workflows*, and *connectors*.
- In a lean app, much of the "heavy lifting" shifts to the platform vendor: data consistency and cross-cloud replication; security and governance; fault tolerance and scaling; infrastructure procurement, deployment, and management; data modeling and API maintenance.
- Developer and ownership benefits include: more secure and predictable outcomes; automatic cross-cloud, cross-company, and cross-region data sharing; guaranteed consistency of data, modern APIs that are cloud-, web-, and mobile-ready, and data model evolution guaranteed not to break clients.
- Companies can adopt lean app methodologies incrementally, and vendors such as Vendia and Snowflake have embraced this model in their offerings already, simplifying migration and adoption.

Progress in Application Design

Software has progressed over time to remove drudgery and toil from the developer by abstracting the level of expression – like assembly language to C to C++ to Java to Python as a programming language example. But it is not just the language used to write applications: The shift from highly proprietary libraries with complex licensing schemes to shared, open source libraries has had a similarly dramatic impact on the speed and efficiency of developing software and delivering IT solutions.

With the move to the cloud, not only did the job of running a data center disappear – the "serverless" moniker is now appearing in virtually every category of service from AWS, another way in which vendors are trying to remove the undifferentiated work of deploying,

scaling, securing, and managing servers and other infrastructure from the job of developing and delivering software implementations.

Watching this grand evolution over the years raises interesting questions:

- Where are *today's* trendlines - increasing automation, abstraction, standardization (de facto as well as de jure), and "serverlessness" - headed?
- What can they tell us about what a typical business software application will look like in the future?
- Most importantly, what can developers, cloud companies, and the broader IT ecosystem do to achieve those future outcomes faster?

While tracking an entire industry is complex, there is a single, prevailing theme to all of these changes: "less is more". Broadly speaking, **progress in application design has been driven by reducing surface area and shifting complexity to the application's environment.**

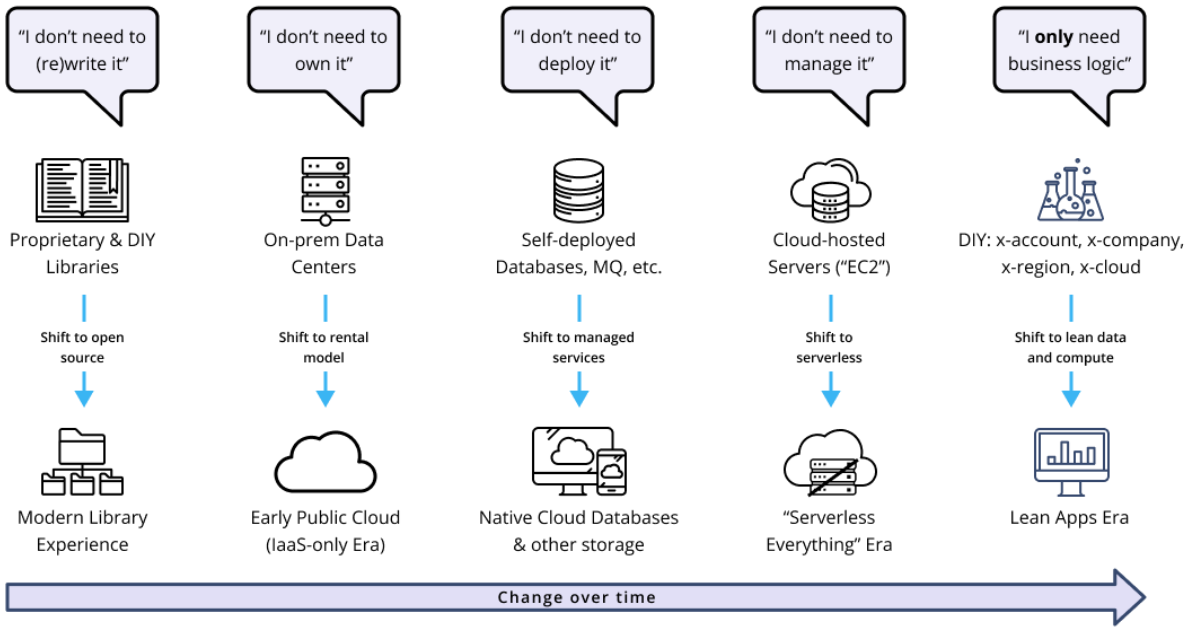


Figure 1: How complexity has shifted out of applications over time through surface area reduction.

Reduction in surface area occurs in various ways:

1. **Removal.** The most powerful "reduction" of all is when an entire area is permanently removed from a developer's purview. The shift from on-prem data centers to the public cloud is a great example of removal. Wholesale problems that used to consume precious IT resources ("Which router should we use?" "What's our network backplane bandwidth between data centers?" "How much rack space do we have left on the west coast?") have vanished … or at least have been condensed into paying a public cloud provider's margins, which are usually covered by the shift from buying peak capacity to paying for actual use.

2. **Reduction through standardization.** When things cannot be taken away entirely from the application stack, the next best thing is to focus a company's (and developer's) resources on the elements that differentiate their customer outcomes, rather than the undifferentiated "commodity" pieces. Open source libraries, the standardization of operating systems and languages, Docker images … all of these are examples where the community has worked hard to avoid spurious differences that consume precious resources, allowing practitioners to focus on what matters. (Note that "standardization" here doesn't have to mean a formal, committee or standards body-driven process; it can also take the form of upvoting open source libraries to create a de facto notion of the 'right' way to handle a common task, such as parsing a URL in python.)

3. **Paradigm shifts that simplify complex tasks.** In the purest form of the above methods, the programming model does not change outright. Rather, tasks like "provisioning a server" or "using a URL parser" still occur, but with less underlying effort (public cloud versus private data center, standard library versus selecting a vendor to negotiate and purchase a custom license). Other times, however, the reduction in surface area requires a corresponding change in how an application itself is constructed.

   This class tends to lead to "epochs" of software development: Like the shift from mainframes to client/server, which eventually created the server-based Unix era, which itself led to the public cloud (initially focused on Infrastructure-as-a-Service), which ultimately gave rise to fully managed and "serverless" offerings. Each of these created powerful new opportunities for application developers, but these large shifts could be achieved *only with a compatible modification in how the software itself*

was developed, organized, and/or deployed. Every paradigm shift also relied on the ones that came before. For example, AWS Lambda was only possible because fully managed storage solutions (such as Amazon S3 and Amazon DynamoDB) had become available, which in turn were only possible because the public cloud (IaaS style) had already been developed.

4. **Automation.** "GitOps", along with commercial vendors like Netlify, and a host of SaaS-style deployment tools have radically changed how most developers approach committing, testing, and deploying code. Twenty years ago, new software projects often started by creating (and staffing) custom tools to perform these functions; now, they are standard, usually SaaS-based, mechanisms, with virtually no thought given to how they actually work, as more of the routine elements are shifted from application developers onto vendors.
   (Since it's orthogonal to the other changes, deployment automation and development tools are not captured in Figure 1.)

Look to the Future

So much for looking in the rearview mirror – what does all this say about the future of software development? If the major trends have been focused on the continual removal of surface area,and the resulting decrease in complexity and cost of ownership, what is the eventual result?

If one imagines these trends continuing, it is also natural to ask an interesting question: *What else can be taken out?* What undifferentiated heavy lifting can be eliminated, leaving only the truest, most pure form of what is often referred to as "business logic"?

In other words, what does a fully lean app look like?

For an industry that has for so long defined "software" as "stuff that runs on a machine", it can be difficult to give up on that paradigm. One approach is to focus on customer value, by asking, "*What are the parts of an application that actually create end user value?*"

For instance, when a developer patches an operating system image, the end users (customers) do not care. At best, it protects them from some future security or operational event, but it is not something they can directly perceive.*They especially do not care who does it* – the infrastructure provider, a third party, the company's own developers, etc. In other words, an activity like updating an operating system image is utterly undifferentiated. A developer patching the OS on a server is not likely to help their company outcompete the competition or enter a new market (unless that company happens to be a cloud provider, operating system producer, or server vendor themselves).

So what is the actual beating heart of a typical application, the part that a company cannot actually forgo? One often hears the term "business logic" tossed around, but what does that actually mean?

The Elements of a Lean App

By letting go of how an application is constructed today and instead thinking about it structurally, there are several elements that embody the actual business needs of a typical application (Figure 2):

- **A data model.** Most commercial applications are, at their heart, "CRUD" in nature – whether buying toilet paper on Amazon, filling out an insurance claim form, or

paying a bill online, much of software is ultimately about reflecting real-world changes with a durable replica (aka “database update”) of those facts, and that requires knowing what kind of data a business needs to model in the first place.

- **Integrity constraints.** Once the “syntax” of a data model is known, the next question is usually establishing the “semantics”, most notably what kind of data is good versus bad data. For example, most US banks will not allow a customer to withdraw more money than they have in your checking account. Thus, in a banking application, attempting a withdrawal would usually start with the software equivalent of asking, “Is the current balance greater than or equal to the withdrawal amount?”

  If not, the withdrawal is not valid. Note, that this does not say how integrity constraints are expressed – they could be data- or logic-centric expressions (balance >= $0.00) or require Turing-complete code to calculate, possibly relying on existing data in the data model to determine the answer.

  Like the data model itself, integrity constraints are pure business logic. Advances in telepathic AI aside, there is no way for a vendor or piece of infrastructure to know what it means for data to be “right” for any given business (though choices in how the data model is expressed – the schema – might make the expression of integrity constraints easier or more difficult).

- **State transitions (aka data triggers).** What do Oracle PL/SQL routines, AWS Lambda functions triggered by an Amazon S3 file upload, cron jobs, and Microsoft Azure LogicApps have in common? They’re all *workflows*.
  They are ways of managing transitions from one application state to another, whether that transition is time based (“run a cron job at midnight”, “retrieve the tweets on a specific subject from the last hour”) or data centric (“when the balance of a checking accounts exceeds $1M, transfer the excess to a savings account”). State transitions might employ integrity constraints, particularly if they create new or modify existing data items, but they are fundamentally *machine-initiated and managed*, rather than representing end user input or output.

  Business workflow modeling is one of the key reasons that software, and thus applications, need to be stateful and Turing-complete, because they cannot be reduced to stateless or trivial pattern representations in most cases.

- **Connectors.** Connectors are the ligaments that enable an app to connect to other applications. Connectors have two conceptual elements, though they may be combined in practice:
    - *Data transformations*. These modulate differences between the app's data model and one or more "foreign" data models.
    - *Event hubs*. Other applications that lack proper eventing, scaling, or buffering mechanisms may require not only data transformation but also *control* integration, ranging from polling (for ingress) to queuing and throttling updates (for egress).

  It is tempting to think of connectors as "impure", often their role will be to make up for shortcomings in legacy applications or systems that exist outside the lean app regime. But even in a perfect world of only lean apps communicating with other lean apps, the need to build and deploy independently ("microservices" versus "monoliths") means that they will need to be loosely coupled, rather than tightly bound to everything. Thus, connectors have an important *long-term* role to play in modulating inter-application schema and workflow evolution, as well as their *short-to-medium term* role in assisting with the initial migration from a legacy app to a lean app and its integration with other legacy systems not yet ready for modernization.

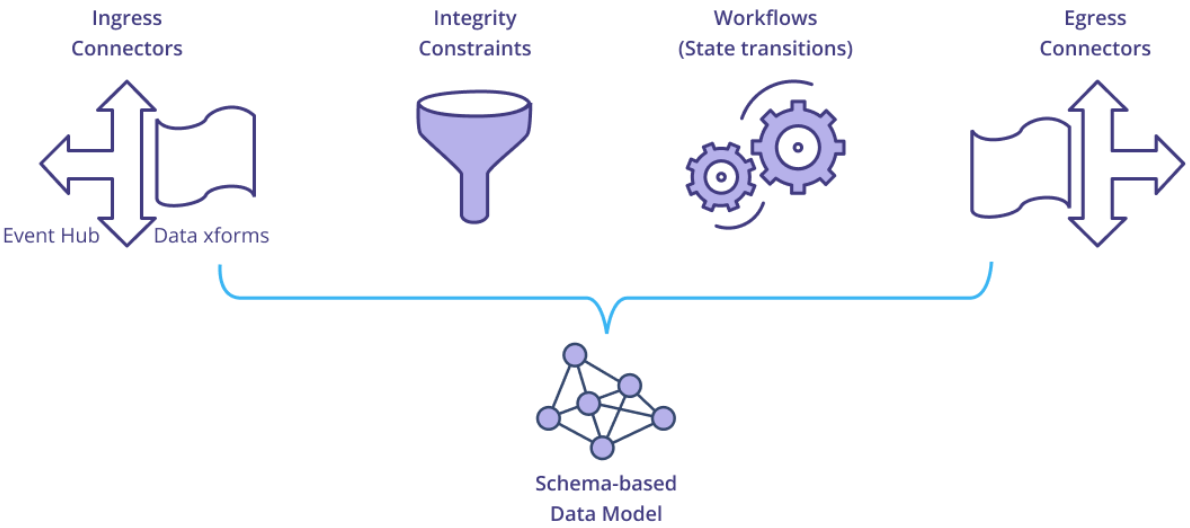


Figure 2: The elements of a lean app.

The list above is short. At first blush it might seem *too* short.

Compared to a "typical" application, such as a Kubernetes-based app, it's perhaps 1-3% of the total amount of software (and deployment/operational) infrastructure normally thought of as an application. And just looking at the list above, even with an eye toward public cloud infrastructure, one would be forgiven for reacting with, "But wait…there's so much more!" And that's certainly true today: There is a critical gap between the list above – true business logic – and the application services developers are using today. To make a lean app possible, *the infrastructure has to start doing more for application developers*. In particular, it needs to do all of the following, automatically and without impacting developers or operators (Figure 3).

There is a gap between true business logic and the application services developers are using today.

Lean apps should be:

- **Schema-based.** Where it makes sense, data models (and their evolution) and integrity constraints should be expressed in the form of a standards-based schema, rather than written into application code or deployment instructions.
  Another way to say this is that "leanness" also extends to preferring data-centric expressions (`BALANCE >= 0`) over code (`if balance - withdrawal_amount < 0 then throw Exception(withdrawal_amount)`). This is to ensure that the expression itself is lean – the less there is to read and write, the less there is to get wrong or have to carefully maintain over time – and to allow for automation over manual effort by shifting constraint enforcement from handwritten code to the platform.

Automating data-driven APIs enables the infrastructure to turn them from "dumb pipes" into "smart, ACID systems".

- **Automated API generation.** Let's face it: Part of what makes existing apps "plump" as opposed to "lean" is actually developer hubris, including an over-reliance on customization and a sense that hand-written code somehow "protects" the application from vendor drift, future requirements, etc. These belief systems fade over time – for example, most developers today could not even imagine writing in assembly language and performing manual machine register allocation, even though those tasks were once commonplace skills.

  Today, API design is often viewed as IP, when in fact many applications do not need a highly customized API so much as they need a clear, easily scaled and secured API that works well and evolves safely.

Automatically generated APIs are not just faster, though - automating data-driven APIs enables the infrastructure to also turn them from “dumb pipes” into “smart, ACID systems” that ensure that the data they transport, even when it spans different companies, can be kept consistent and up to date at all times.

- **Automated schema and API migration.** Business needs are not static, and therefore application data models can not be “one and done”, either. So while lean apps strive for minimalism in their design, they do need to embrace the reality of a business’s long-term ownership needs, which includes ever-evolving customer, market, and internal demands on the software.

  If every such change required rewiring the fundamental structure of the application, it would fail to be lean. (In other words, “leanness” is not just about what is not present on any given day, but also about *minimizing effort over the entire lifecycle of an application*.) And these updates need to affect APIs in a controlled fashion as well, so that both “frontend” and “backend” developers aren’t constantly in the (undifferentiated) business of inventing, applying, securing, and deploying incremental API enhancements.

- **Data decentralization and distribution.** One of the ironies of the public cloud is that many of the most business-critical needs are the hardest to develop for. Building an application that can operate resiliently and safely across multiple regions (with different accounts in each region as a best practice) while also seamlessly ensuring that data shared with business partners is always correct, complete, and up to date is one of the hardest challenges in software systems today.
  Only the most elaborate financial systems and a few mega tech companies can also do cross-cloud support and have the ability to resiliently remain available when a major cloud provider goes down. To succeed teams must shift the complexity of maintaining ACID transactions into the infrastructure, leaving the application developer with the simpler problem of deciding what data updates have to happen as a single transaction versus separately (since that grouping is an element of the application’s semantics, and can’t be guessed by the infrastructure).

Everything else - cost-optimized cross-cloud data transport, data integrity and security, high-speed replication with ACID properties, etc. - should be handled by the infrastructure.

- **Built-in access controls and privacy protection.** If a company needs to "roll their own" access controls in order to share data with partners, other departments, or different applications, then their infrastructure is not working hard enough. These elements should be part and parcel of the data modeling services provided by the application platform.

Everything else - transport, data integrity, security, replication with ACID properties - should be handled by the infrastructure.

- **Ledgering and versioning data.** The data model should implicitly support versioning and ledgering of data, rendering the need to manually maintain logs, backups, or audit controls unnecessary.

- **Cross-cloud.** Applications, and application developers, can not afford to pretend that other clouds do not exist. Business partners, mergers and acquisitions, and other routine events limit even the most "all in" company from ignoring other cloud vendors apart from their preferred choice.
  And increasingly, picking the best-of-breed cloud services means reaching across providers to select different aspects. Doing that inevitably requires an application infrastructure that offers *cross-cloud data and code capabilities as an intrinsic feature*. This stands in sharp contrast to the Kubernetes, "port it to every cloud and run it yourself there" approach, which is costly in terms of time, people, and infrastructure spend, and still results in monocloud silos that are tough to interoperate.

- **Scalable, pay-per-request, SaaS-style infrastructure.** Whether one calls it "serverless", "fully managed", "PaaS", or something else, application infrastructure needs to be as easy to consume, scale, and operate as adding another user to a Slack account. The days of an already overburdened application team being willing to deploy, scale, monitor, and maintain another company's software are gone forever, and the expectation is clearly that infrastructure should be "adult", as in, not require babysitting to scale, secure, maintain, or otherwise keep it running as intended.

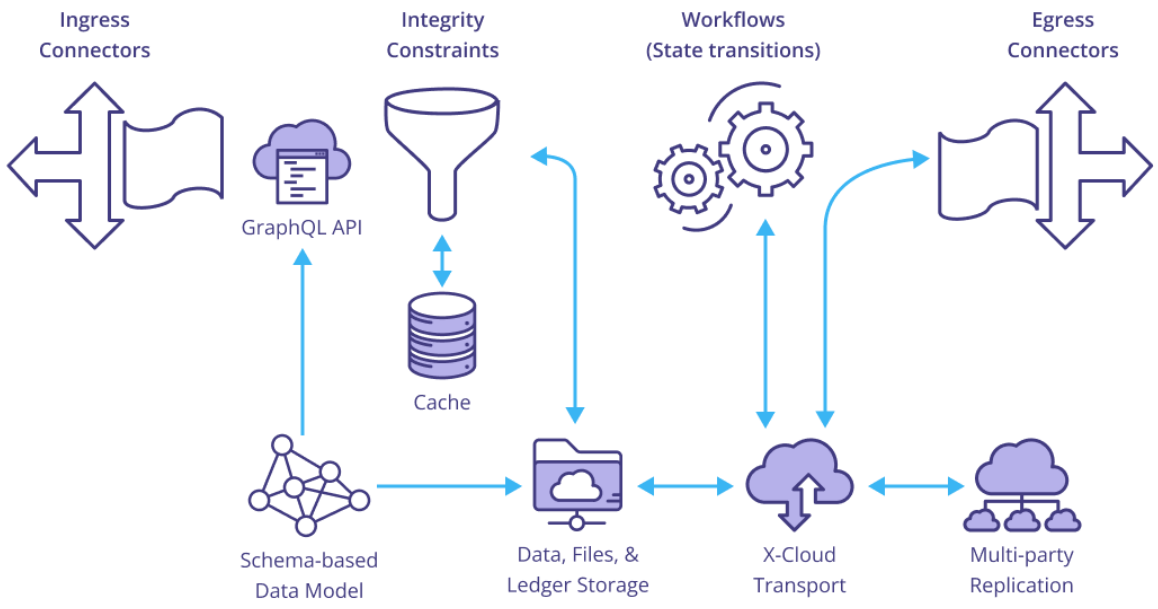


Figure 3: Key capabilities of a lean app that are provided by the platform.

Example: Acme Bank

Imagine a simple example based on Acme Bank, which has been a commercial bank in the past but now wants to get into retail banking as well and needs an application to model its new retail banking needs, such as ATM and teller-based deposits and withdrawals for end user checking accounts. The discussion below ignores some obvious business details (like savings versus checking accounts) to keep things simple.

Start by expressing a very basic data model:

- User_Account(customer:string, balance:number)

There also needs to be a way to express which cloud providers, regions, and parties need to share the data modeled above. Say that Acme itself wants to operate in two regions, both on AWS, to satisfy regulations that require fault tolerance in the event of a region-wide outage, but also needs to share data with a regulator who operates in the Azure cloud:

- Acme_East_Region(cloud:AWS, region:us-east-1)
- Acme_West_Region(cloud:AWS, region:us-west-2)
- FINRA(cloud:Azure, region:eastern_us)

Access controls are fairly simple: Acme can read and write balances from either region, but FINRA can only read balances, since it's just there to ensure compliance with banking laws.

Then invent some notation:

- User_Account(read:{Acme_East, Acme_West, FINRA}, write:{Acme_East, Acme_West})

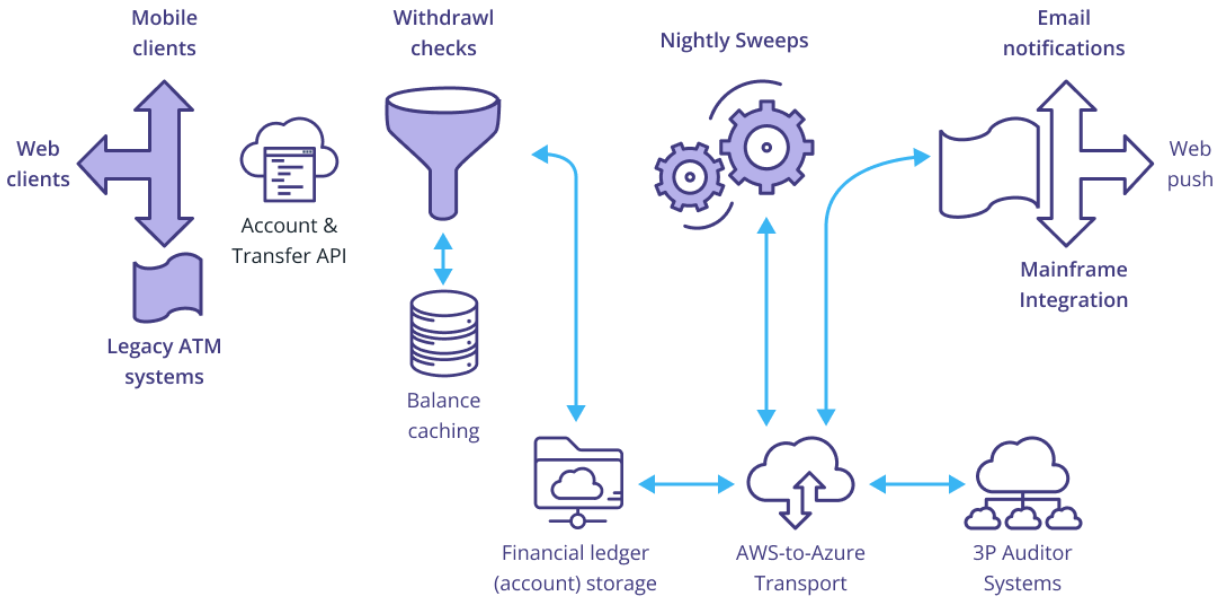
Note: In a more elaborate example, these settings might need to vary on a per-item basis based on the actual data; here, they are expressed at a per-node level as part of the overall data schema as partner-level data sharing rules that don't vary from account to account.

Then add some simple integrity constraints:

- User_Account.balance >= 0
- User_Account.name IS UNIQUE

Because Acme has other systems that need to know about account updates – including legacy systems that run on a mainframe – create a simple event hook to notify them of changes our app has processed:

- EventHub.webhook(<*mainframe's API URL*>[1])



[1] If this isn't a public API, it might require adding a fourth node to enable event delivery through a VPC/VPN integration back into Acme's on-prem mainframe or other legacy data center systems.

Figure 4: How a lean app for a retail banking solution might look.

From here, infrastructure can supply everything else that i's needed by the application:

- Cross-cloud replication of all financial accounts, deposits, and withdrawals.
- Mobile- and web-ready GraphQL APIs generated from the data model that can be connected to end user experiences without further implementation work.
- User transactions ("ACID" reads and writes) that enable complex financial transactions to be grouped together without race conditions.
- Versioning, lineage, and logging of all updates, including "audit-ready" capabilities that allow a third party auditor to be attached to the system in a read-only manner.
- Infrastructure deployment, scaling, monitoring, and managing of the entire banking application.
- Full security apparatus, from on-the-wire and at-rest data encryption to PCI and SOC2-compliant access controls and governance mechanisms that is compliant out of the box.
- Machine, network, cloud provider, and geographic fault tolerance to meet demanding financial service industry requirements.
- Multi-party decentralization with tamperproofing of all data, regardless of size, type, or ownership that allows for sharing anti-laundering information with regulators, central banks, and federal authorities.

Notably, what is *not* present is anything related to infrastructure: servers, Kubernetes management, database deployments, etc. It is truly lean – *the vast majority of what goes into a typical application is gone*. Even though this lean application can do many things that even the best and brightest hand-written "conventional" applications can not do today, including cross-cloud, cross-region, cross-party data sharing with ACID guarantees through APIs that are guaranteed to evolve in a secure and backward-compatible fashion over time.

Of course, this example is not fully reflective of all the real-world needs of an application. In practice, it might need a transformation between an existing web app ("front end") and the generated GraphQL APIs, custom workflows to look for unusual balances or activity that run overnight as batch processes, additional event handlers to hook up other legacy systems and applications, and so forth.

But the central idea, *that much of the manual labor of replicating data and crafting "data APIs" can be removed and replaced with smarter infrastructure solutions*, remains. Figure 4 illustrates the architecture of our lean banking app.

Adoption Concerns

Lean apps offer significant benefits over traditional application design approaches. But like any significant change, care is required to avoid pitfalls.

Common concerns include:

- **"Training wheels" that won't scale with production usage.** A persistent problem, especially with purely "client-side" tools and frameworks is that they are ultimately just training wheels; once a practitioner becomes fluent in the underlying system or platform, the need for the assistance diminishes.
  Low-code solutions, especially front end ones, also tend to have growing pains: by limiting what is easy to express, they can also limit what is possible to accomplish. By contrast, lean apps are implemented in the same way as robust, production-grade public cloud services: they offer scalability through multi-tenancy and tend to show off some of their best ROI as applications scale up in both usage and complexity. And the ability of lean apps to offer consistent solutions to some of the most pressing problems (cross-cloud data sharing, decentralized single source of truth with partners, etc.) means that many companies and developers may actually need some time to "grow into" their full range of capabilities.

- **Containers and Kubernetes.** Kubernetes is a powerful, but unfortunately, also very complex, technology. It is also structurally disposed towards the lowest common denominator of the public cloud, which tends to deflate the value that businesses can get from public cloud adoption.

  Lean apps offer a different thesis: that best-of-breed public cloud services are important to leverage, but companies (and developers) need help connecting them to their data (and other systems) and retaining the ability to change those decisions over time. However, it is also important to realize that containers are not the opposite of lean apps – in fact, they are one of the best ways to express many of the lean app business logic mechanisms, such as integrity constraints and workflows. And even if a developer has an existing solution that cannot be ported away from a Kubernetes or server-based implementation initially, there are two ways to retain it: by treating it as a microservice that *connects* to lean apps, or by using it as the stateful workflow engine *within* a lean app. The latter approach typically limits some of the serverless benefits, cost improvements, and scalability of the lean app, but may still provide a useful incremental step to acquire the many other benefits that a lean app provides.

- **Cost.** Understandably, cost is a dominant concern any time IT vendors are being considered or new approaches are being introduced. Lean apps offer significant cost savings over traditional approaches because they are built on serverless principles, meaning they "scale down to zero." hen no work is being performed, they cost nothing to operate. That tight cost enveloping means that teams do not need to scale to peak capacity, as is usually required for server- or container-based solutions. Lean apps also shift critical but difficult cost optimizations, such as cross-cloud and cross-region data routing, data storage lifecycle management, and large object storage transmission and replication, to the platform, allowing them to be optimized with best-of-breed solutions instead of by an already strapped developer just focused on delivering the basics for a time-sensitive project.

- **Lock-in.** One of the most critical aspects of lean apps is that *they diminish reliance on a single public cloud provider:* by virtue of being able to add new partners, regions, and cloud nodes with just a couple lines of configuration, vendors like Vendia, who specialize in lean app methodologies, help customers break geographic and cloud service provider dependencies with every app they build or port. And since lean apps focus on standards-based artifacts (container images, JSON/JSON Schema, GraphQL, HTTP) they keep developers from having to learn proprietary languages, file formats, or approaches.

Are Lean Apps a New Idea?

Of course not – like most changes, the concept of lean apps is just an evolution of what is come before, weaving established trend lines together and giving a name and strategic direction to inevitable processes that are nascent today, but growing fast.

"Serverless" cloud services, for instance, captured some of the essential ideas of Lan Apps, but missed out on the idea that the data model is essential to truly simplifying application development.

Cloud-based data lake companies like Snowflake made the leap of treating cross-cloud as a feature, rather than a porting exercise left to the reader, a la Kubernetes, but remain focused on specific analytics and BI-based solutions rather than the more general problem of application construction. Cloud-based databases, such as Google's Spanner, have started down the road of supporting cross-region solutions, but have yet to fully embrace the idea of cross-account (let alone cross-cloud or cross-party) data as a built-in feature.

Blockchains, such as Hyperledger Fabric and Ethereum, embody the idea of distributed data models that can span companies, clouds, and technology stacks, but are missing the scalability, performance, fault tolerance, cloud integration, and application code support that would be necessary to host typical IT business solutions, especially those with serious privacy, compliance, and scalability requirements. Open source software achieves "zero marginal cost" sharing, but is not a solution for delivering SaaS-style operations without manually hosting, scaling, and managing it all.

By bringing all these elements together, companies like Vendia make lean apps possible – offering an application framework that enables developers to express, deploy, and operate a lean app *today*.

The Lean App Movement

The Lean App Goal: Flip the IT Iceberg

Every company talks about innovating to serve customers. Every developer wants to spend their time on the activities and outcomes that matter (the tip of the iceberg), not pointless, repetitive drudgery (the much larger underwater portion). And yet, modern application development across every country and industry sector is dominated by undifferentiated activities.

What would it take to "flip the iceberg" – to get to a place where 90% or more of a team's time, energy, and infrastructure spend goes towards innovation and competitive differentiation, and less than 10% is spent on babysitting infrastructure, repetitive development and deployment tasks, and other undifferentiated heavy lifting?

Key to this company and team transformation is an associated application transformation: **moving from a style of application development that depends on a broad, expensive surface area to one where the surface area is as minimal as possible – with more work and more complexity shifted to infrastructure vendors.**

A Lean App Manifesto

What does that world look like? In some ways, it is a continuation of everything that's come before: open source software, the public cloud, Serverless. Each of these helped to "lean out" some aspect of application development, without getting quite all the way there.

The lean app movement is driven by a simple, overarching idea: Less over more.

If a technology, process, approach, or vendor can do something effectively that is not an organization's core business, then let them. Outsourcing work is a blessing, and helps avoid Tyranny of Choice mistakes that are unfortunately common among developers who mistake "more control" for "better outcomes over time". Code, infrastructure, and tools are costly, long-term liabilities – *remove what you can, standardize what remains*.

The lean app movement is driven by a simple, overarching idea: Less over more.

Specifically for applications, lean apps follow 6 simple rules:

1. **Differentiated business logic over undifferentiated commodity code and activities.** If it is not unique to a company, or does not matter to the end users, why spend the company's time, money, and people to do it?

2. **Smart APIs over "dumb pipes".** If an application has to poll repeatedly for data or expend code (and developer effort) dealing with missing, out-of-date, or inconsistent information, then it is a dumb API problem. Smart APIs enable teams to concentrate on application concerns, not the details of how data is secured, replicated, shipped across clouds, or made ACID.

3. **Schemas over code.** If there is an easy, standards-based way to express something about data model or its integrity constraints (initially or over time), use that, in preference to writing (and maintaining) expensive, custom code to do the same thing.

4. **Automation over manual effort.** API generation, schema evolution, transactional data replication, event generation…virtually every application development team struggles to recreate these elements, and yet none of them are differentiated ("business logic") outcomes.

5. **Off-the-shelf data sharing and data security over costly "DIY" data and security coding.** If your application is reinventing the wheel by trying to solve cross-company, cross-cloud, cross-region, or cross-account data sharing (with a focus on security,

scalability, cost management, fault tolerance, and compliant data access controls), then something is wrong.
DIYing any *of these data sharing solutions is the poster child of undifferentiated heavy lifting.* Lean apps embrace consistent, platform-provided security, compliance, and governance, rather than leaving these as application-specific coding requirements to be performed over and over again with every new project.

6. **SaaS over IaaS; server*less* over server*ful*.** Deploying or operating another company's software? Manually scaling systems up and down that should be "adult enough" not to require you to babysit them? Are not AWS/Azure/GCP yet spending time and money to keep servers or other infrastructure healthy? If end users do not benefit directly from an activity, perhaps time is better spent innovating elsewhere.

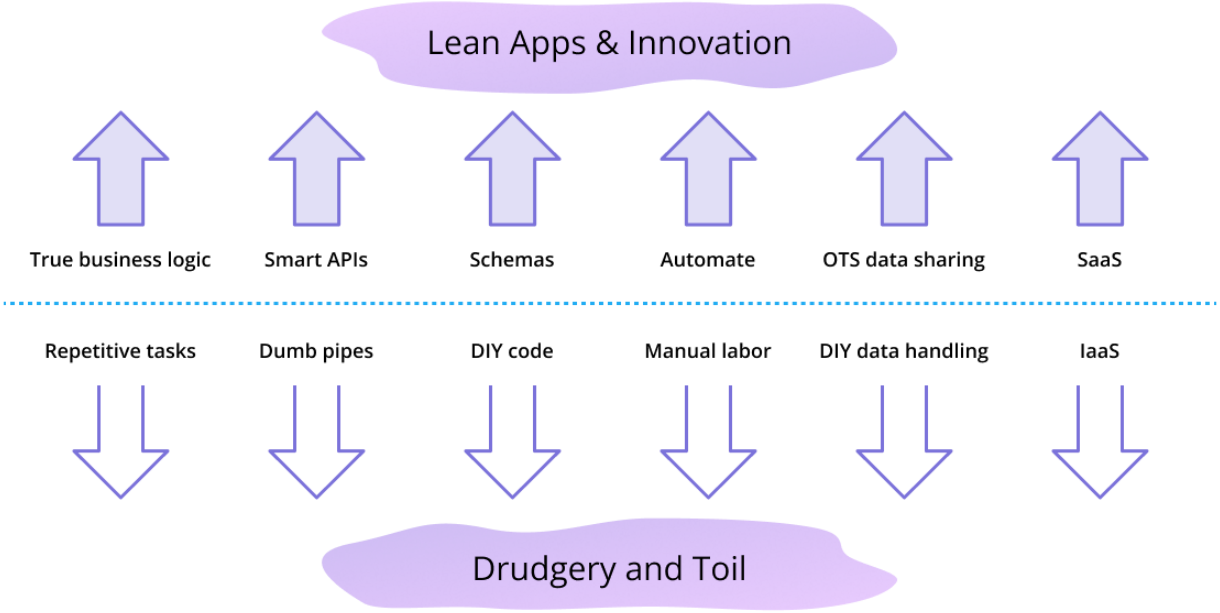


Figure 5: The six pillars of lean.

Getting Started

Is it possible to create lean apps today? What about "brown field" applications that can not be discarded and recreated overnight … can they benefit from lean ideas or incremental migration strategies?

Like any significant movement, change takes time: despite the transformative effect of the public cloud, the majority of business applications still run on-prem. In fact, only in 2020 did cloud spend outstrip on-premise IT spending for the first time in history. For many

companies, a quarter century or more of mainframe and on-prem hardware and software investments mean that they still have a long way to go to fully adopt the cloud.

Serverless solutions – such as AWS's Lambda and Fargate – while representing some of the fastest growing elements of that cloud provider's portfolio, still represent only a fraction of total compute spend on the AWS platform. Lean apps will follow similar lines, if for no reason other than the fact that they also benefit from public cloud adoption and further development of serverless offerings.

Working on a green field application or feature? Companies like Vendia will let you move directly to a lean app methodology for new application development or to layer a significant new feature or capability on top of an existing application. And where wholesale adoption of the idea is not (yet) possible, developers and companies still benefit from the concept through incremental steps in various areas of application development:

APIs

- Adopting schema-based solutions for APIs (such as the Open API specification, formerly known as "Swagger") and data.
- Prefer GraphQL over REST APIs when possible, to future-proof both client and server interfaces. This also enables adopting a lean app-style schema evolution approach, even if the schema evolution needs to be "manual" for now rather than automatically computed and applied.

Data Storage

- Adopting cloud-based databases, especially "serverless" offerings, such as Amazon Aurora Serverless, Amazon DynamoDB (with the serverless scaling/pricing option enabled) or Google Spanner.
- Utilize built-in data capabilities from cloud service providers where possible, such as automated backup solutions or cross-region data replication (but realize read consistency models typically degrade when using the latter).

Application Design

- Segregating integrity constraints and workflow code from "infrastructure management" code, making it easier to migrate off the latter over time.

- Adopting fully managed cloud-based services where possible for data storage, event hubs, queuing, API management, etc. This will make it easier to adopt a lean application framework over time.
- Prefer serverless options over “serverful” ones where complex application state or data manipulation is not required.
- Prefer standardized server (machine) or container images where possible over custom solutions.

Final Thoughts

The concept of “leanness” has been around for ages – in a sense, it’s just DRY coding applied at a macroscopic level to every aspect of an application, from data handling to cross-cloud deployment.

But as the next major iteration of software development, it will once again transform the industry much as the cloud has, shifting ever more work from individual developers, their applications, and the companies that pay their wages, onto cloud and other providers who will create economies of scale through multi-tenanted solutions that handle the undifferentiated (if still exceeding complex) heavy lifting of data and compute management. Over time, companies will enjoy lower TCO, faster time to market, and better ROI as the process of developing, deploying, and maintaining software applications and associated IT infrastructure continues to drop.

Lean coding is the ultimate expression of innovating for customers.

About the Author

Dr. Tim Wagner, the “Father of Serverless,” is the inventor and leader responsible for bringing AWS Lambda to market. He has also been an operational leader for the largest US-regulated fleet of distributed ledgers while VP at Coinbase, where he oversaw billions in real-time transactions. Dr. Wagner co-founded Vendia with Shruthi Rao in 2020 and serves as its CEO and Chief Product Visionary. Vendia’s mission – to help organizations of all sizes easily share data and build applications that span companies, clouds, and geographies – is his passion, and he speaks and publishes frequently on topics ranging from serverless to distributed ledgers.

linkedin.com/in/timawagner
t: @timallenwagner
www.vendia.com/blog