Evolutionary Architecture: 24 Principles of Emergent, Organic, and Highly Adaptive Design

Ar·chi·tec·ture (*är'kĭ-tĕk'chər*) is the fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution [IEEE]. From this classical definition, we get the sense that an architecture is the broad overall framework, scaffolding, support structure, and foundation for a computer software system or application. We even see terms like principles for guiding its design and evolution. Where we get in trouble is when we apply or superimpose this definition on a global system of system, ecosystem, enterprise, large solution, or simply a complex system. That is, we attempt to define a very complex set of components and their internal and external interrelationships. Another weakness of this classical definition is when we apply traditional thinking principles like linear waterfall or staged lifecycles, synthesize or fabricate too many system requirements well-in advance, or attempt to create an ironclad architecture that lasts years or decades. In this case, an extended system architecture phase is applied, a large architecture is synthesized, cost and quality become an issue, and of course complexity and testability are challenged. The length of a traditional system lifecycle is often years and decades, few of these programs or projects actually materialize, and when they do, customers are often dissatisfied, the market requirements have changed, technological solutions are obsolete, and the cost of change is prohibitive.

Ev·o·lu·tio·nar·y Ar·chi·tec·ture (*ĕv'ə-lo͞o'shə-nĕr'ē • är'kĭ-tĕk'chər*) is a set of principles to incrementally define, implement, evaluate and improve extremely small, simple, modular, replaceable, useful, and valuable components, systems, and ecosystems. From this we assume that market or customer requirements exist as tacit inexpressible knowledge, a small number of needs must be solutioned quickly, tangible end user feedback must be gathered fast, and this cycle is repeated until optimal value is achieved. Each iteration is so fast it is economically feasible, any design or implementation may be fully replaced at any time, and the goal is to converge upon an optimal design after multiple short attempts. With this process, elastic, flexible, or extensible ecosystems can be organically grown, expanded, or contracted as needed. Enterprises do not need to create large designs over years or decades, cost, risk, and failure are reduced, technological obsolescence is embraced, and early business and market value is realized. Entrepreneurial startups depend upon these principles out of economic necessity, they often succeed quickly if they keep the scope and size of their solutions small and fast, and they often bring innovatively new products and services to market quickly. However, not everyone embraces evolutionary architecture principles, they are often counterintuitive, capital intensive enterprises often attempt large, multi-billion-dollar, multi-decade architecture and design solutions, and few people can simply think small.

Although the principles of evolutionary architecture and design themselves have been known, evolved, and applied with some success over the last 40 or 50 years, only recent technological breakthroughs have enabled their full application and realization. The advent of the electronic computer in the 1940s and computer programming languages in the 1950s were the fundamental building blocks of evolutionary architecture. However, the managers and engineers who inherited these 20th century technologies often applied 19th century linear project management techniques to their creation. Some of these 19th century ideas included cost and schedule planning, extensive business and systems analysis techniques, and the creation of decade long architectures. Thus, in spite of having some of the most malleable, flexible, and elastic technologies known to humankind, managers and engineers sought to treat software applications like they were tangible rigid structures, components, and matter. This, of course, was a major mistake. Although computer scientists knew that the software was an intangible medium akin to psychological thoughts, it wasn't until the late 20th and early 21st century that social psychology could be applied to the development of computer programs. That is, customer needs exist as inexpressible thoughts that must incrementally and gradually be teased out of their minds, which was made possible by the software industry (1970), personal computer (1980), world wide web (1990), and smartphones (2000).

Early thought leaders in this field were definitely Frederick Brooks, Edsger Dijkstra, David Parnas, and even Gerald Weinberg, who taught us about the malleability, structure, modularity, and psychology of software. Of course, who can forget Victor Basili (iterative development), Tom Gilb (evolutionary development), and Barry Boehm (spiral development), who taught us about gradual elaboration of software. Early methodologies like Joint Application Design (JAD), Rapid Application Development (JAD), or even Participatory Design (PD) taught us about rapid design solutioning with real customers and end users. However, the clear thought leaders were the creators of agile methods, such as Ken Schwaber and Jeff Sutherland (Scrum), Kent Beck (Extreme Programming), and Martin Fowler (Refactoring), along with the purveyors of early evolutionary architecture principles (Neal Ford and James Shore), who taught us about a sharp focus on collaboration, teamwork, working software, fast feedback, system replaceability, and business value. Probably, the sharpest thinkers were the purveyors of lean thinking principles, such as James Womack, Mary and Tom Poppendieck, Dave Anderson, Wally Hopp, Ed Pound, Don Reinertsen, and Dean Leffingwell. As a synergy of early thought leaders, lean and agile thinking, and emerging technologies such as microservices, containers, cloud computing, mobile computing, DevOps, application security, and UX, let us explore some of these timeless principles together.

1. **Mar·ket • Driv·en • In·i·ti·a·tion** (*mär'kĭt • drĭv'ən • ĭn-ĭ-sh'ē-ā'shən*) Ask, need, want, necessitate; To make an urgent request

   - ✓ **Pull-based system**
   - ✓ **Non-predictive planning**
   - ✓ **Demand-based value stream**

   Of course, all architectures, design activities, and end-results should be the result of a direct market request or need. That is, organizations should establish enterprise value streams from which market or customer needs are made. Once a customer or market request is made, then, and only then should a design-making activity ensue. There may need to be an initial triage to determine the scope, size, and magnitude of the request. However, requests should not be arbitrary, the firm's value stream should be pre-primed with the types of requests that are allowable by customer or market representatives. You don't ask for a boat from a car dealer, and you don't ask for a live whale from a seafood restaurant. Typically, there are menu choices that are provided. Once a customer or market representative has selected a product or service from the enterprise, and/or its

value stream, then the firm should determine whether it has enough capacity to fulfill that service, the service becomes backlogged, or even rerouted to another product or service provider. Remember, the goal is to render the product or service with the shortest lead and cycle time, so customer or market requests should not be ignored, backlogged indefinitely, or simply forgotten. It's okay to let customers know in-advance whether the enterprise can take that order at all. It's okay to indicate the enterprise no longer accepts certain types of requests. Once an order is placed, the customer should be notified when to expect the result, the work should be scheduled, and the customer should be given regular updates, including final delivery. The only work enterprises should initiate apart from customer or market requests should be to add new products and services to their offerings. However, even these should be based upon some valid quantitative or qualitative market data.

2. **Ec·o·nom·i·cal·ly • Fea·si·ble** (*ĕk′ə-nŏm′ĭ-kəl′ə-lē • fē′zə-bəl*) Low-cost, modest, affordable; To make something inexpensively

- ✓ **Inexpensive work items**
- ✓ **Affordable scope and size**
- ✓ **Relatively low cost and effort**

A triage team should size, scope, and estimate the effort, cost, and duration of a customer or market need for a new product or service once it has been logged into an enterprise's value stream as a pull or demand-based request. Its overall economic feasibility must be immediately determined based on the policies of the value stream, work in process (WIP) limits, capacity, materials, resources, and other considerations. The characteristics or attributes of an order for a new product or service must not exceed the predetermined constraints of the value stream's operating performance. If the value stream has been designed for a work item of a particular type, kind, scope, or magnitude, then the customer or market request must conform to these constraints. It's possible that an enterprise value stream may have some flexibility to shift some resources to adapt to a particular customer or market request, but workload balancing must be reasonable and should not exceed the general operating parameters. For instance, let's say a team can work on two or three work item of a particular type or kind at one time and there is a need for four or five. The enterprise may elect to shift some resources away from a team with excess capacity to take on the extra two or three items. However, let's say that any team is not sized to design a product or service requiring more than 20 to 40 hours' worth of effort. In that case, a customer request for 200 to 400 hours or more should not be accepted, because to do so would certainly exceed the operating characteristics of a team's workflow. Remember, the goal of a lean value stream is the shortest possible lead, wait, and cycle time along with overall workflow balance from end-to-end. A value stream must always have some built-in excess capacity to account for workflow balancing and creative endeavors.

3. **Two-Piz·za • Team • Read·y** (*to͞o-pēt′sə • tēm • rĕd′ē*) Pair, duo, couple, crew, squad, group; To form a very small response team

- ✓ **Two-person teams ideal**
- ✓ **Small tiger team acceptable**
- ✓ **One-or-two-week cycle times**

An appropriate number of people for a small design group its what's known as a two-pizza team. That is, the entire team can be fed with approximately two pizzas at a time. If you know anything about a programmer's appetite, that isn't too many people at all. Ideally, each team should be bigger than one person, target a couple of pair programmers, or three or four people if necessary. Occasionally, a small tiger team may need to be slightly larger depending on the scope of the customer or market need, request, or work-item-type. A good number for a large team is Miller's limit, or seven plus or minus two people for a team taking on a larger minimum viable product (MVP). In today's DevOps intensive environment, 70% to 80% of your changes should be very small changes, improvements, or variations of existing code or designs. That is, requests consist of improving performance, fixing a bug, or making a very small functional design improvement. In this case, only a duo of pair programmers should handle each request, they should only take a few minutes, and small groups of pair programmers in larger organizations should be able to make 50 to 150 deployments per day, perhaps 10 times more in the case of very large organizations. For new designs or significant modifications or enhancements team size and duration may be slightly longer, perhaps one to three days. However, even these teams should be able to make dozens of code commits per day in the form of incremental design improvements until the final code deployment. In the case of an entirely new MVP for a new customer or market need, or to explore or validate new concepts, a team of seven to nine people should take no more than 10 business days. It may be necessary to execute two or three two-week design sprints to appropriately scope an innovatively new MVP.

4. **Just-in-Time • E·lab·o·ra·tion** (*jŭst-ĭn-tīm • ĭ-lăb′ə-rā′shən*) Fast, quick, prompt, punctual; To immediately assemble a solution

- ✓ **No predictive activity**
- ✓ **Request based solutioning**
- ✓ **Address immediate work item**

All new designs should be initiated based on a new, triaged, validated, and approved request. Pair programmers or small teams should self-organize and self-select the next work item type based on capacity, resources, expertise, and need. Work should never be forced onto a team, teams should not be preselected by managers, and more work should not be allocated to a team's queue that would otherwise exceed their WIP limits. It's possible a team may be taking longer than expected with a problematic work item type, resources may need to be shifted away, or life happens a team suddenly loses its predictive capacity. In that case, WIP limits may need to be lowered, the work item type delayed or canceled, or the request may need to be placed on another queue. Once the team has self-selected and formed, the request is pulled from the queue, the request is clarified with the product owner, product manager, customer, or end user, and any needed advice or resources are sought, then design work may ensue. No design work commences until all of these conditions have been met, enterprises should not be tasking teams with out-of-scope design work that is not related to a specific customer request, and teams should not be designing in anticipation of a request. That being said, not all customers are market or external end-users, some customers

may be internal customers. Perhaps an innovation team is exploring a new MVP or a marketing representative wants to showcase a new technology at a trade show. However, all work that comes into the value stream must be entered through a formal request to the value stream, not a personal request. Furthermore, it is acceptable for a team to use planned or excess capacity to refactor, reduce technical debt, upgrade its platform, try out a new technology, or explore an innovative solution.

5. **Ut·ter • Sim·plic·i·ty** (*ŭt'ər • sĭm-plĭs'ĭ-tē*) Easy, plain, basic, bare, minimal, unadorned; To plan an uncomplicated solution

- ✓ **Simplest design possible**
- ✓ **No excess functionality built**
- ✓ **Fulfill exactly one requirement**

Each design should be the simplest design possible, only to address the specific needs, requirements, or attributes that pertain to each customer or market request. The ultimate goal of each team is to fulfill the customer or market request with the fastest lead and cycle time and shortest wait time. Therefore, the simplest design solution should be sought for each customer need. In the case of existing design patterns, it may be necessary to strip down a design to its essential elements or create entirely new simple designs to satisfy each need. No excess capacity, attributes, features, functions, or other attributes should be added to the design other than what is absolutely necessary. Once again, 70% to 80% of the customer or market requests should be to enhance, fix, or modify an existing product. In the best case, a new request can be fulfilled with an existing design pattern that does not need to be created from scratch, provided there is no excess design capacity in that pattern. For instance, a customer may request a shopping basket for an ecommerce website. The design firm should already have a shopping basket design pattern established that can be shipped lock, stock, and barrel. Or, perhaps, an existing shopping basket design pattern merely requires a small modification or enhancement. Through simplicity, attributes such as quality, performance, security, reliability, and dependability can be assured, because there are little to no unknowns in the design of the solution. The design should be fit for purpose and fulfill the exactly one attribute, feature, or function if possible, versus a myriad or plethora of unrelated characteristics. Product owners and product managers should be trained, taught, and skilled at scoping work item types with the smallest number of attributes as possible and should not be making over scoped requests.

6. **Quan·tum·ly • Small** (*kwŏn'təm-lē • smôl*) Tiny, little, miniature, atomic, molecular; To create the smallest possible solution

- ✓ **Atomically small design**
- ✓ **Bare bones size and scope**
- ✓ **The smaller the size the better**

Likewise, each design solution should not only have a simple design but should also be quantumly small as well. We've stressed a number of times that 70% to 80% of requests upon teams should be minor enhancements, fixes, or modifications. That is, most requests should not consist of entirely new designs, but rather design variations. Each request should not only be simple, but as small as possible. If lead, cycle, and wait times are to be fast, short, and minimal, then each request has to be as small as possible. By making requests throughout the value stream fit within the capacity of each team, within the WIP limits, and within other resource, scope, and design constraints, enterprises can ensure the overall enterprise value stream is well-balanced. Remember, unlike traditional thinking enterprises, portfolios, programs, projects, the teams, the goal is to eat the elephant one bite at a time rather than swallowing the elephant, filling teams to full capacity, and over allocating teams. Exceeding capacity expands or extends lead, cycle, and wait times to infinity and is never a good idea. Although it seems intuitive or instinctual to fill teams to full capacity to ensure the enterprise is getting its money's worth from its investment in resources, they're actually ensuring enterprise failure by doing so. By keeping each request and design as quantumly small as possible, teams have plenty of built in capacity to properly solution requests, they can focus on the smallest possible design to satisfy the customer or market needs, and they can ensure attributes such as quality, performance, reliability, and speed are satisfied. Small designs lead to superior quality, customer satisfaction, delivery speed, workflow performance, and overall satisfaction of stakeholders at all levels including customers, end-users, executives, managers, and the delivery team too.

7. **Ar·chi·tec·tur·al·ly • Min·i·ma·lis·tic** (*är'kĭ-tĕk'chər-ə-lē • mĭn'ə-mə-lĭs'tĭk*) Littlest, slender, smallest, least, slightest, nominal; To create the simplest architecture necessary

- ✓ **Architect only for now**
- ✓ **Don't architect for future**
- ✓ **Minimally acceptable design**

Sometimes, a customer or a market request for an innovatively new product or service may be a little more complex than a simple design enhancement, fix, or modification. That is, a work item type may require more effort than a simple design solution, but rather a more complex architecture as in the case of a sophisticated globally scaled microservice. However, even in the case of a complex global, enterprise, or market application or microservice, it's architecture should begin as a smaller tightly scoped MVP that should take no more than a couple of two-week design sprints for a small executive tiger team to specify. The implementation team itself can then incrementally evolve a simple architectural solution over multiple two-or-three-week sprints not to exceed 90 days or an annual quarter. The design team of course, should not attempt a big bang architecture at the end of the quarter, but incrementally evolve the architecture or design, ideally with code builds every few days. Some of these smaller code releases may even be deployed to product managers or lead users in the form of splash screens, UX wireframes, or simple system functions. Like a smaller work item type requiring a few hours or a few days of simple design solutioning, the architecture team should strive for the simplest architecture possible. Remember, this isn't a year-long system architecture or a decade-long enterprise architecture or system of systems. This is only an MVP, so the architecture team should think small, lean, slim, slender, minimal, and necessary. Again, like a small design team, this is a just-enough, just-in-time architecture only to satisfy the needs, constraints, and attributes of an MVP, gather market data,

rinse, and repeat. Since MVPs only have a short shelf-life, teams should not assume solutions last longer than 90 days.

8. **Or·gan·i·za·tion·al • In·de·pen·dent** (*ôr'gə-nĭ-zā'shən'l • ĭn'dĭ-pĕn'dənt*) Free, separate, liberated, autonomous, unconstrained; To operate independently of organizational design

   - ✓ **Don't automate hierarchy**
   - ✓ **Organizationally free designs**
   - ✓ **Organizations should adopt design**

   Historically, organizations often task architecture and design teams to codify the workflow of a given organizational hierarchy. Architecture and design teams are often chartered or formed within an organizational department or element, so they are often tasked to implement solutions to automate the operations of that organizational element. Thus, system architectures and designs reflect a given organizational hierarchy. This is always a bad thing for a number of reasons. For one, organizations are inefficient bureaucracies not high-performance microservices like mobile apps. Imagine if Amazon's online bookstore automated the workflow of a brick-and-mortar bookstore how horrible that would be? Another major reason not to automate an inefficient organizational bureaucracy's hierarchy, is because they constantly reorganize every 90 days with promotions, demotions, and other shifts in political power. So, if the architecture team is on a 10-year project building a decade long architecture, then the team has to recreate the system architecture every 90 days over a 10-year period. That's about 40 architectures over a decade. If the architecture team manages to fly under the radar for a decade, then its architecture is 10 years out of date or about 40 times removed from reality. Therefore, the goal of an architecture is to satisfy what is needed vs. how to design it, with a focus on time-to-market, simplicity, smallness, speed, quality, efficiency, and value. That's everything but a 10-year architecture to codify an inefficient organizational bureaucracy. Architecture teams need to focus on usability, user experience, performance, convenience, and value, not codifying a mountain of obsolete multi-million-dollar business requirements documents written a decade ago. An architecture's goal is to satisfy a market need, not a political bureaucracy.

9. **Just • E·nough • Waste·free • De·sign** (*jŭst • ĭ-nŭf' • wāst-frē • dĭ-zīn'*) Capable, functional, useful, powerful, strong, productive; To efficiently and effectively produce intended result

   - ✓ **Smallest design possible**
   - ✓ **No excess capacity or margin**
   - ✓ **Only solution for immediate need**

   Each architecture and design should focus on satisfying the needs of exactly one customer or market request or need, or attributes of an MVP from an executive design sprint. Once again, most architectures and designs will be small enough to solution with a simple enhancement, fix, or modification. In fact, each new enhancement should focus on refactoring the existing architecture or design to remove excess waste unseen or undetected by prior teams. Architecture and design teams, like most humans, tend to exhibit unusually high degrees of cognitive blindness. That is, their ugly baby is always beautiful, even though, in fact it is ugly. In technical terms, architecture and design teams tend to over scope their solutions by as much as 80% to 90% or more no matter how hard they try. It takes real skill and expertise to create the smallest, simplest, and most minimalistic architecture and design possible. Probably less than 1.5% of all architects and designers have this skill. The rest of us suffer from over scoping disease. Therefore, there's always an opportunity to slim down the architecture and design, strip out the excess waste, and slim down its attributes to the minimum necessary to fulfill the customer or market needs. Since physical structures are not malleable, flexible, nor easy to change, often times hardware teams will build in excess capacity for years, decades, and even centuries. Many military weapon systems are engineered for 50, 75, or more years. However, software is highly malleable and the economics of expanding and contracting its architecture and design are negligible. Therefore, software in particular lends itself to under scoping an architecture, leaving out details subject to technological obsolescence, refactoring or removing suddenly discovered waste, and extending them later if necessary.

10. **Ob·ject • Based • Mod·u·la·ri·za·tion** (*ŏb'jĭkt • bāst • mŏj'ə-lə-rī-zā'shən*) Parts, units, items, piece, block, element, ingredient; To compose a solution as a small highly-interchangeable component

    - ✓ **Object oriented design**
    - ✓ **Hide implementation detail**
    - ✓ **High-cohesion, loose coupling**

    To the greatest degree possible, each team should apply object-oriented principles to their architectures and designs. That is, having well-published, exposed, and public class interfaces in form or methods and hidden or encapsulated design and implementation details that are not exposed to consumers. In doing so, build system, application, or microservice designs as a collection, set, family, or ecosystem of modular building blocks not unlike highly modularized Legos or Ikea furniture. In other words, architects and designers can compose larger and more complex solutions from smaller building blocks with well defined interfaces, modify or extend the overall end-to-end solution with ease, or simply remove what they do not need. This method of composing architectures or designs allows teams to minimize dependencies between system functions, features, and attributes, ensure system quality, reliability, and resiliency, and even apply proven design patterns over and over again. In the case of microservice architectures, the web or mobile app itself may be an object-oriented module or building block that can be added or subtracted to or from the application ecosystem to enhance overall user experience. That is, enhance value adding microservices or remove inconvenient microservices that don't enhance usability or ease of use. In the case of modern information technologies, methods and data are often exposed in the form of application programming interface (API) calls or classes are inverted to expose their functions and capabilities. In the case of refactoring, often times a single object may be split into two modules or two modules may be combined into one while stripping out excess waste in order to simplify the code, remove dead code, and enhance quality, reliability, performance, and security among other non-functional attributes.

11. **Eas·i·ly • Im·ple·ment·a·ble** (*ē'zə-lē • ĭm'plə-mənt'ə-bəl*) Effortlessly, painlessly, leisurely, undemandingly, straightforwardly; To perform a task with the least amount of effort

- ✓ **Minutes for small designs**
- ✓ **Day or two for medium designs**
- ✓ **Week or two for innovative designs**

Among all of the attributes discussed so far, perhaps easy implementation is the key to good architecture and design. That is, a customer or market request or work item type to an enterprise value stream should easily fit within the capacity of design teams. As we've alluded to over and over again, the majority of the designs should be in the form of tiny enhancements, fixes, and modifications that a small team can implement in minutes, resulting in hundreds of market or customer deployments per day. However, as we've also suggested, good architecture and design take expert skill and not everyone has the skill to make appropriate design solutions. Therefore, it's imperative to have maximum transparency, teach and reward good architecture and design choices, and foster a culture of openness, mastery, continuous improvement, and willingness to fail if necessary. Can an individual architecture or design choice break a build? It sure can! But, then the DevOps pipeline should revert to a previously known, safe, and operational state, the entire architecture and design team should chip in to help, effort should be rewarded not punished, and everyone should learn from challenges. More junior architects and designers may be assigned nominal enhancements, fixes, and modifications, and more senior personnel may be assigned more complex tasks. However, T-skills should be promulgated, and senior personnel should be paired with more junior personnel to enhance their skills, level the playing field, balance the team and enterprise workflow, prepare for nominal attrition, and develop personnel at all levels. If the scope of customer requests is small, work item types fit within WIP limits and team capacity, the scope of architectures and designs is kept to a minimum, and T-skills are developed to broaden skills, then easy implementation is possible.

12. **Fail·ure • Free • Qual·i·ty** (*fāl'yər • frē • kwŏl'ĭ-tē*) Solid, stable, strong, reliable; To create a solution that satisfies market needs

- ✓ **Form test criteria first**
- ✓ **Develop tests before design**
- ✓ **Immediately test implementation**

Of course, teams should sharply focus on creating the highest quality architectures and designs possible. That is, solutions that satisfy market or customer functional and non-functional needs as well. Functional requirements, elements, or attributes should be kept to a bare minimum, because complex solutions, architectures, and designs mask quality issues, while simple solutions easily satisfy quality constraints. Furthermore, non-functional requirements, elements, attributes, and acceptance criteria should also be specified as part of customer or market needs, work item types, epics, features, user stories, and MVPs. These, of course, should also be kept to a minimum as well. Information security, usability, and overall performance should be prioritized, and experts in these non-functional attributes should be consulted from the beginning, if not included in the team. Non-functional requirements should be baked or built into architectures and designs, rather than iced on or boot strapped on as an afterthought. If the enterprise or team is applying good architecture and design patterns, then the non-functional attributes or characteristics of those patterns should be known in terms of defects, failures, usability, security, capacity, scalability, and system performance among many others. In today's Internet of Things (IoT) where microservices are scaled to billions of global end users at once, information security architectures and designs should also be a forethought rather than an afterthought. The use of containerized microservices are helpful, good information security design principles, and automated vulnerability testing, monitoring, and remediation. The quality attributes of simple architectures and designs are easier to manage, control, and ensure, while, vice versa, the quality attributes of complex solutions are simply uncertain.

13. **Ex·treme·ly • Fast** (*ik ˈstrēm'lē • făst*) Speed, velocity, quickness, acceleration; To create an extremely high-performing solution

- ✓ **Design for high performance**
- ✓ **Fewest decision gates possible**
- ✓ **Fastest response times possible**

System or application performance of architecture and design solutions, like information security, usability, and reliability, is paramount to the success of modern web apps, microservices, and market facing products and services. Modern applications are consumed by billions of users at one time, so the highest system performance is a critical goal or objective of today's architecture and design teams. System performance is not just critical due to the sheer number of end-users a single web or mobile app may encounter, but system performance matters even to a single or small number of users. Remember, with lean startup techniques, the attributes of an MVP may be tested on a small group of lead users, so poor system performance may have significant effects on go, no-go decisions concerning innovatively new products and services. Today's end-users are extremely sensitive minor variations in system performance, a split-second matters in today's IoT, and a single system delay can lead to the attrition of thousands, millions, and even billions of users, while a split second increase in performance may instantly garner millions or billions of users. Simple architectures and designs are the key to superior system performance and solution teams should be sensitive to the performance of their implementations. Again, like information security, system performance attributes need to be documented as acceptance criteria, automated system performance tests created and run, and system performance experts, architectures, and designs selected and applied. Sometimes, architectures and designs need to be refactored, stripped down, denormalized, and inverted in order to enhance system performance to the greatest degree and measure possible. Network and ecosystem performance may also impact performance of individual applications.

14. **De·light·ful·ly • Use·ful** (*dĭ-līt'fə-lē • yo͞os'fəl*) Handy, helpful, pleasing, enjoyable; To create an unusually convenient solution

- ✓ **Design for usability**
- ✓ **Easier and fun to use**

✓ **Perform vital functions**

Application usability is sort of the last frontier of architects and designers. Solution teams, largely due to cognitive blindness, often think in terms of sequence, hierarchy, and structural architecture and design elegance. However, today's end users are desperate for the most usable designs and delightful user experience. Applications with bad usability and UX come a dime a dozen and few applications are engineered for the global IoT marketplace with world class levels of usability and UX. Google and Amazon are two of tens of thousands of application designers whose products relentlessly focus on delightful usability. MVP teams often evolve applications to have less and less amounts of usability and UX. With each MVP cycle, web and mobile apps contain more and more features, more and more sequences and screens, and more and more menus and buttons. For instance, let's take a mobile weather app. The customer or end user just wants to see the current temperature, chance of rain, and other critical weather conditions at their location. Don't make them login, verify, get a special verification code, navigate to the temperature screen, watch an ad, ask them for their location, and then drill down two more screens to find the temperature. A mobile phone knows its location, it should simply tell the user the current temperature, high for the day, chance of rain, when its going to rain, and other critical weather attributes in fractions of a second without watching an advertisement for new clothing. Perhaps another term for usability and UX is convenient, make your mobile and web apps convenient, intuitive, and instinctual for the simplest user. Extend ecosystems to provide a helpful app rather than add a new attribute to an existing app that decrements usability and UX. Simplicity leads to delightfully useful, while complexity does not.

15. **Baked·in • Se·cu·ri·ty** (*bākt-ĭn • sĭ-kyo͝or'ĭ-tē*) Safe, assure, guard, preserve; To proactively build-in high-assurance protection

✓ **Establish security criteria**
✓ **Apply high-security design**
✓ **Test for vulnerabilities early**

We've already alluded to the necessity of building in security into today's market, customer, and end-user facing mobile and web apps. Again, information security and even privacy are super critical in today's global IoT ecosystem. Each customer or market need or exploratory MVP should come with a set of security and privacy requirements and should not be treated as an afterthought and simply bootstrapped on in a superficial manner. Teams should be trained and certified in information security and privacy principles, security architecture and design patterns applied, requirements specified, procedures followed, subject matter experts present, and security and privacy tests applied, run, and passed. Furthermore, using modern containerized microservices, security and privacy should be firewalled off from the rest of the ecosystem. That is, mobile or web apps should not be able to violate your privacy and security, and vulnerabilities within your microkernel or application code should not be able to peek out of your container's walls. Most operating systems come with 40 to 50 high priority common vulnerabilities and exposures (CVEs), so using microkernels minimizes the number of CVEs held within your container. Furthermore, containers minimize penetration of these vulnerabilities from inside or outside your container. The platform ecosystem itself should have vulnerability monitoring, your container should have its own internal vulnerability monitoring, and enterprises should monitor the status of their deployed microservices as well. Microservices should be immediately terminated upon detection of a vulnerability without impacting the overall health, status, and operation of the rest of the ecosystem. For instance, any web or mobile app can stop without crashing your device. Better security is possible with simple architectures and designs.

16. **Au·to·mat·i·cal·ly • Test·a·ble** (*ô'tə-măt'ĭ-kə-lē • tĕst'ə-bəl*) Verify, check, validate, evaluate, assess, appraise, ensure, assure; To create a solution that can be reliably confirmed to satisfy objective criteria

✓ **Establish automatic tests early**
✓ **Automatic test upon code commit**
✓ **Functional and non-functional tests**

Solution teams should create their architectures and designs to be automatically testable. Acceptance and test criteria should accompany each MPV, customer or market request, and epic, feature, and user story. Automatic tests should be created first and ready to run upon implementation. Code should be incrementally implemented to satisfy architecture and design constraints. As the code elements are checked into version control, tests should be automatically executed, and solution teams notified of results. Non-functional code attributes or characteristics should also be reported such as size, complexity, performance, and security. All code in the version control system should be kept error free, all tests should pass, and only code that passes unit tests should be integrated, delivered, and deployed. Independent test teams may contribute additional unit, component, integration, system, acceptance, performance, certification, accreditation, and deployment tests that must pass. Solution teams should create their architectures and designs for testability. Tests should also include coverage analysis to ensure there is no excess, dead, wasteful, or unreached code. Testing is not a natural skill and should be taught, rewarded, and encouraged. Testable architecture and design patterns should be established and known. It's not just about architecture and design elegance, that is, what looks good to the designer, but good architecture and design is also about what performs good to the system tester. A good architecture and design is a testable one, a testable one is a simple one, a simple one is a deployable one, and a deployable one is a solution that can be delivered to a customer or end user for feedback. The ultimate judge of quality is the end user who cannot provide any valid market feedback if code can't be tested and delivered.

17. **Quick·ly • De·ploy·a·ble** (*ˈkwĭk-lē • dĭ-ploi'ə-bəl*) Send, provide, supply, distribute; To automatically deliver solutions to market

✓ **Design code with end-state in-mind**
✓ **Run all deployment tests upon commit**
✓ **Automatically deploy code if tests passed**

Architectures and designs should result in code that is quickly deployable. All of this is dependent upon feasibility, simplicity, smallness, minimalism, efficiency, modularization, ease, quality, performance, usability, security, and testability. Customer or

market needs exist as tacit knowledge, tangible explicit knowledge cannot be known until real end users have code in their hands, and the value of MVP attributes cannot be determined without this feedback. This is why any architecture that takes more than a few weeks, months, years, or decades is worthless. The only way to determine the market value of an architecture and design is to put its implementation into the hands of real users as soon as possible. With this data and information feedback in-hand, enterprises, decision makers, managers, technical experts, and solution teams can refactor their architectures and designs to enhance any lacking attributes or sharpen good ones. The ultimate test of an architecture or design is its business value and the only way to determine that is whether customers or end-users perceive your applications to be of value. The only way to determine that value proposition is to deploy code to them. Therefore, solution teams must create their architectures and designs with quick deployment in-mind. A deployable solution is a successful one and a non-deployable solution, well, is indeterminate, or worthless in the worst case. An ultimate measure of success is not lines of code, sprint velocity, estimating accuracy, utilization, architecture or design pattern purity, size or cost of team, length of project, number of tests, nor delivery to staging. The ultimate measure of success is number and speed of deployments to real end-users to measure market or customer feedback. Of course, even this can be gamed if deployments are not valuable.

18. **Con·tai·ne·rized • Mi·cro·ser·vic·es** (*kən-tā′nə-rīzd′ • mī′krō-sûr′vĭs′əs*) Software, application, tool, utility, web or mobile app; To produce a small self-contained application software module

- ✓ **Design for containerization**
- ✓ **Each application self-contained**
- ✓ **Each app contains micro-kernel OS**

Containerized microservices are one of the latest innovations enabling high degrees of emergent architecture and design feasibility. It's possible to have a microservice without a container. However, when used together, they form a powerful combination or one-two punch for satisfying many of the architecture and design principles we've identified so far. These include, but are not limited to feasibility, simplicity, smallness, minimalism, efficiency, modularization, ease, quality, performance, usability, security, testability, and deployability. A microservice is really a very small, atomic, quantum, and self-contained market-facing application that provides immense business value to enterprises, customers, and end-users. When deployed within containers, microservices may operate independently of an enterprise infrastructure, such as on a mobile device, including tablets, smart phones, wearables, and other embedded devices including smart appliances and even automobiles. Containerization even provides a degree of information security and privacy. Basically, containerized microservices are complete self-contained virtual computers. Computers within computers that can perform valuable functions for their ecosystem in spite of their ecosystem. There are some dependencies for special platforms such as iOS, Android, Windows, and other platforms. However, these self-contained application environments provide the ability to create micro architectures, designs, and customer, market, and end user functions that provide very useful features. These may include mobile texting, emailing, calling, searching, maps, social networking, banking, insurance, financial services, entertainment, and a myriad of other functions not previously possible in PC applications with complex 100 million line of code bases.

19. **In·de·pen·dent·ly • Val·u·a·ble** (*ĭn′dĭ-pĕn′dənt-lē • văl′yo͞o-ə-bəl*) Useful, helpful, worth, benefit, critical, necessary, important; To create a service or product solution of immense tangible or intangible utility

- ✓ **Solutions should be valuable**
- ✓ **Each should perform vital function**
- ✓ **Each has tangible and intangible worth**

Each solution team must strive to create an architecture or design that results in an application that is independently valuable. In agile terms, these are called feature teams which create a vertical slice of end-user facing functionality, rather than a shared service team that creates a horizontal layer like hardware, network, operating system, middleware, etc. Therefore, architecture and design teams are essentially feature teams creating vertical slices of end-user facing functions rather than horizontal intermediate solutions. That is, the goal of each team is to create an end-user facing application, no matter how small or narrowly sliced. This could be a login screen, single sign-on identity access management capability, data or information query, performance report, enterprise monitor or tracker, or ability to submit essential data into the enterprise or system. If you were designing an automobile, this might be a keyless entry, remote start capability, accelerator, speedometer, fuel or battery indicator, global positioning system, map, communication device, entertainment unit, collision avoidance system, etc. Each of one of these components has a valuable end-user facing function. Contrast these to tires, suspension, wiring, unibody frame, or other infrastructure or platform component. Therefore, the purpose of each MVP, epic, feature, or user story should result in a vertical architecture and design slice that will perform a vital end-user-facing function. Oftentimes, a work item type may be to remove a defect, improve system performance, enhance usability or security, or increase convenience. With software, and especially containerized microservices, this enables solution teams to create architectures and solutions that can be quickly deployed to millions and sometimes billions of global end users to enhance value.

20. **One·click • Us·a·bil·i·ty** (*wŭn-klĭk • yo͞o′zə-bĭl′ĭ-tē*) Simple, obvious, natural, intuitive; To create a very easy-to-use solution

- ✓ **Don't make user apply multiple operations**
- ✓ **Don't bury operations under multiple layers**
- ✓ **Don't inundate user with excess information**

One-click usability is a variation of delightfully useful that focuses on utter simplicity of the ultimate end-user experience upon which architecture and design teams strive. Today's applications, especially mobile apps are not about comprehensiveness, architecture and design complexity and hierarchy, and inundating and assaulting users with data and information on every square inch or centimeter of display space. Furthermore, architecture and design teams must not create applications with

multiple, hierarchical levels of buttons, menus, and screens to access critical functions. Perhaps, it's a misunderstanding among architects and designers who believe that the critical function is a complex hierarchy of functional choices, like providing a 50-page restaurant menu with 200 items from 17 different countries. Today's end users need instant access to critical functions, data, and information, such as find their current location, ability to send and receive information, order a product or service, or be entertained. End users don't want to navigate through 10 clicks and 5 screens to find out the current temperature, humidity, and chance of rain. This speaks to almost every evolutionary and emergent design principle we've discussed so far, such as feasibility, simplicity, smallness, minimalism, efficiency, modularization, ease, quality, performance, usability, security, testability, deployability, and value. Architects and designers need to focus on a single attribute of immense end user value, create the simplest, smallest, inexpensive, secure, fastest, and easiest to use solution, and deliver a tested high-quality solution quickly. Your application should often perform one function and do it well. Simplicity is a lost art, is sometimes more complex than complexity itself, and embodies the principles of architecture and design minimalism.

21. **In·stant·ly • Re·place·a·ble** (*ĭn'stənt-lē • rĭ-plās'ə-bəl*) Temporary, swap, transferable, expendable, alterable, exchangeable; To create an automatically interchangeable solution component

- ✓ **Design and code for replaceability**
- ✓ **Frequently refactor design and code**
- ✓ **No impact for continuous replacement**

Solution teams must also create simple, minimalistic single-function architectures and designs that are instantly replaceable, not by you competitors, but by themselves. Part of this has to do with quick deployability, but more so, it is about continuous improvement, refactoring, performance and security, and enhancing the user experience itself. Much of it has to do with modularity, testability, and the resiliency of containerized microservices. Ultimately, it has to do with continuously optimizing the value of the product, service, customer and market satisfaction and delight, and of course internal and external business value. System or application development is not a destination, but a journey, and end-user applications must be continuously enhanced, improved, tested, and redeployed with increasing frequency without disrupting the operation of the end-user's ecosystem, platform, or mobile device. This can only be realized if the solution or application is architected or designed for replaceability. This can only take place if it is simple, small, minimal, object oriented, and based on an architecture and design of containerized microservices. Achieve these architecture and design goals, and enterprises can instantly replace their applications with such frequency and even transparency as to continuously enhance user experience and value without interruption. Think of a common search engine, texting app, email app, weather app, or entertainment app. These can be enhanced or replaced hundreds of times per week while you operate your device. Every time an end user starts an app, they can conceivably be using a new version that has been entirely enhanced and replaced. Instantly replaceable is the one of the ultimate goals of today's architect and designer, that is, to continuously refactor, enhance, improve, and redeploy solutions.

22. **E·co·sys·tem • Friend·ly** (*ē'kō-sĭs'təm • frĕnd'lē*) Group, family, collection, community; To create set of compatible components

- ✓ **Components should enhance ecosystem**
- ✓ **Ecosystem provides great user experience**
- ✓ **Provide rich component variety in ecosystem**

If architecture and design teams are going to continuously refactor, enhance, improve, and redeploy solutions hundreds of times per week, their solution apps better be ecosystem friendly. That is, they should seamless fit into an ecosystem without degrading the user experience of the end-user's ecosystem. Solution teams should create architectures and designs for applications that enhance the user experience of the ecosystem. That is, fill in a gap, needed product or service, or enhance the user experience beyond what is currently available. Either fill in gaps, enhance weak links in the user experience value chain, or provide an entirely new chain of value adding microservices that enhance the overall user experience. Evolutionary architecture and design principles allow solution teams not only the ability to create innovatively new single function components, but then create a series of single function components that enhance the value chain. With over 3 million Android apps and 2 million iOS apps, its hard for today's architects and designers to imagine what gaps exist or how to reach 4 billion global end-users flooded with information overload to present a new app with an enhanced user experience. Part of the solution is to assemble some small lead user or focus groups, put your MVP in front of them and gather their feedback on your new architecture and design solution. Although it seems ludicrous, it doesn't take much imagination to see the gaps and shortfalls in the ecosystems of most mobile platforms. Today's end users often use their mobile devices in their automobiles, drive fast, and need to make fast and safe decisions in fractions of a second. Place yourself in an automobile, drive 60 mph, cover 30 miles in 30 minutes non-stop, and ask yourself how to make better quality of life decisions in fractions of a second.

23. **En·vi·ron·men·tal·ly • Re·sil·ient** (*ĕn-vī'rən-mĕn'tl-lē • rĭ-zĭl'yənt*) Flexible, elastic, pliable, extensible, survivable, buoyancy; To create a solution that can co-exist and survive in a dynamically changing environment

- ✓ **Design components for resiliency**
- ✓ **Components insensitive to environment**
- ✓ **Components shouldn't degrade environment**

Of course, while your solution teams are striving to apply sound principles of evolutionary architecture and design principles, the creators of other apps on an end-user's ecosystem may not. End-user platforms are simply chaos and applications are colliding like asteroids amalgamating into planets orbiting a young star. Furthermore, hackers are exploiting vulnerabilities in networks, operating systems, platforms, and the apps themselves with increasing veracity. An end-user's ecosystem platform is like a battlefield and your application may be the next victim. Therefore, solution teams need to focus upon architectures and designs that are environmentally resilient. That is, they must withstand the constant collisions and demands for

processing speed, memory, network bandwidth, platform capacity, and even security and privacy vulnerabilities. It's sort of like driving around in a Humvee while wearing a flack jacket to buy a gallon of milk from the grocery store while asteroids are falling on you. In other words, your architecture, design, and application must survive the constant collisions and warfare going on in the IoT. Architectures, designs, and applications simply cannot be brittle or subject to instant failure at the first collision. Simplicity, minimalism, performance, security, privacy, reliability, and quality are key. Each architecture, design, and application must focus on a single function and do it well, while exhibiting ironclad resiliency to failure. Sounds like we're advocating a new field of architecture and design resiliency engineering. Yes, there are a lot of demands on today's architects and designers. It's not just about architectural and design pattern purity, knowing the latest pattern design language or representation, or being a certified architect or designer. Today's solution teams need to build platform resilient applications.

24. **Prod·uct • Or·i·ent·ed • Pur·pose** (*prŏd'əkt • ôr'ē-ənt'ĭd • pûr'pəs*) Wares, goods, services, commodity, merchandise, saleable; To develop a solution destined for an end-user, community, or market

- ✓ **Design components for market use**
- ✓ **Don't design components for projects**
- ✓ **Design component with end-user in-mind**

We'll close with a quick treatise on product-oriented purpose. We've alluded to this in almost every architecture and design principle covered so far. Architecture and design solutions must begin with a customer request, they must be small and simple, they must be easy and implemented fast, they must be high quality and testable, and they must be deployed quickly to gather feedback. Furthermore, they must be independently valuable, delightfully usable, secure, resilient, and utterly simple—One-click usability! All of this leads to each deployment resulting in a market, customer, or end-user facing product of great business value to the enterprise and customers themselves. More importantly, architectures and designs must result in applications that are organizational independent, meaning they don't automate inefficient bureaucracies, but perform vital functions for real market end-users. This all flows nicely into today's lean thinking model that urges architects and designers to establish products vs. projects or long-lived, market-facing, demand-based, end-to-end value streams. Each requirement of that value stream is a customer request or work item type that is pulled into the enterprise value stream based on available capacity, resources, and materials. Work item types are architecturally small, so they can be solutioned, tested, and deployed quickly with high quality applications. Contrast this with the traditional thinking model that invests millions of dollars in business analysts to amass thousands of business requirements representing imaginary customer requirements that are pushed onto helpless and hapless under-skilled development teams. While over-allocating and overutilizing these teams, traditional designers are creating 10-year enterprise and system of system architectures that will be obsolete in 90 days.

Summary

So, what exactly have we learned by this short and impromptu treatise on evolutionary architecture and design principles? Well, we've learned that planning, architecture, design, and even testing have always been necessary elements of good application design and some measure of discipline will always be necessary. We've also learned that software is not similar to its hardware counterpart, it is flexible, malleable, and changeable, and we've learned that software is a great medium in which to economically tease out tacit customer needs. We've also learned that lean and agile thinking principles as they've evolved and matured over the last 40 or 50 years are a better disciplined alternative to traditional linear thinking that results in large volumes of business requirements, decade long architectures, and high program and project failure rates. More importantly, we've learned that today's technology stack in the form of the Internet, World Wide Web, Cloud Computing, Mobile Computing, Containerized Microservices, and DevSecOps Pipelines, is the ideal foundation upon which to apply disciplined lean and agile thinking to create market facing products. This combination or synergy of disciplined evolution, software flexibility, and automated verification, validation, and delivery enables enterprises, programs, projects, and teams to quickly field small architectures and designs to build innovatively new products and services of immense business value, user experience, security, and dependability for global consumers.

22 Principles of Successful Evolutionary and Emergent Architecture and Design

1. **Market Driven Initiation**—Pull-based system; Non-predictive planning; Demand-based value stream
2. **Economically Feasible**—Inexpensive work items; Affordable scope and size; Relatively low cost and effort
3. **Two-Pizza Team Ready**—Two-person teams ideal; Small tiger team acceptable; One-or-two-week cycle times
4. **Just-in-Time Elaboration**—No predictive activity; Request based solutioning; Address immediate work item
5. **Utter Simplicity**—Simplest design possible; No excess functionality built; Fulfill exactly one requirement
6. **Quantumly Small**—Atomically small design; Bare bones size and scope; The smaller the size the better
7. **Architecturally Minimalistic**—Architect only for now; Don't architect for future; Minimally acceptable design
8. **Organizational Independent**—Don't automate hierarchy; Organizationally free designs; Organizations should adopt design
9. **Just Enough Wastefree Design**—Smallest design possible; No excess capacity or margin; Solution for immediate need
10. **Object Based Modularization**—Object oriented design; Hide implementation detail; High-cohesion, loose coupling
11. **Easily Implementable**—Minutes for small designs; Day or two for medium designs; Week or two for innovative designs
12. **Failure Free Quality**—Form test criteria first; Develop tests before design; Immediately test implementation
13. **Extremely Fast**—Design for high performance; Fewest decision gates possible; Fastest response times possible
14. **Delightfully Useful**—Design for usability; Easier and fun to use; Perform vital functions
15. **Bakedin Security**—Establish security criteria; Apply high-security design; Test for vulnerabilities early
16. **Automatically Testable**—Establish automatic tests early; Automatic test upon commit; Functional and non-functional tests
17. **Quickly Deployable**—Design code with end-state in-mind; Run deployment tests upon commit; Automatically deploy code
18. **Containerized Microservices**—Design for containerization; Each application self-contained; App contains micro-kernel OS

19. **Independently Valuable**—Solutions should be valuable; Each perform vital function; Each has tangible and intangible worth
20. **Oneclick Usability**—Don't make user apply multiple buried operations; Don't inundate user with excess information
21. **Instantly Replaceable**—Design and code for replaceability; Frequently refactor design and code; No impact for replacement
22. **Ecosystem Friendly**—Components should enhance ecosystem; Provide great UX; Provide rich ecosystem variety
23. **Environmentally Resilient**—Design components for resiliency; Insensitive to environment; Don't degrade environment
24. **Product Oriented Purpose**—Design components for market use; Don't design for projects; Design with end-user in-mind

We've also learned that traditional thinking principles from the 19th and 20th centuries are not, nor will they ever be, the correct approach for rapidly developing flexible software applications on the medium we call the Internet, WWW, and Mobile IoT platform. Creating decade long, multi-billion-dollar portfolios, enterprise architectures, programs, projects, and systems of systems is the surest way to failure as the data reveal. There is simply too much uncertainty in customer and market needs, desires, problems, patterns, cycles, market conditions, political economies, rules, regulations, laws, and constantly changing technology alternatives. An integrated master schedule, enterprise architecture, business requirements specification, system of system design, program or project team, budget, linear lifecycle, rigid process, document suite, governance board, and time are the enemies of good enough. These process elements are simply a façade, model, representation, facsimile, WIP, and waste that do not provide any real value, especially when they are completely fabricated by managers, engineers, and analysts independently of customer collaboration. Even if a customer feedback is sought, the pure volume of requirements, activities, tasks, people, resources, and design elements will lead to so much uncertainty, risk, and complexity that they will most certainly kill your software project dead on sight like Raid. Solution complexity, in spite of its glamour, elegance, cost, size, and length, is simply not the answer in today's IT marketplace.

What's the answer to successfully architecting and designing software and information technology-based end user applications? Keep the architecture and design market driven, inexpensive, simple, small, minimal, waste free, modular, easy, high quality, fast, delightful, secure, testable, and deployable. More importantly, apply containerized microservices architectures and technologies, DevOpsSec pipelines and tools, and rapidly deploy, improve, replace, and evolve valuable market facing products and services. The KISS principle has never been more applicable than it is when applying evolutionary architecture and design principles for creating innovatively new products and services in today's global IoT-driven marketplace. Smaller is better, faster, and cheaper, and bigger is worse, slower, and more expensive. Time, cost, and complexity are a modern enterprise's enemies, not just in terms of the unmanageable uncertainty they cause to delivery teams, but to enterprise survivability and resilience itself in today's highly competitive global marketplace. If size, cost, and time are exactly what large enterprises have, and conversely, what the smallest enterprises and highly motivated entrepreneurial teams to not have, then there is nothing left except market erosion and failure for large enterprises. That is, unless one can teach the elephant to dance and behave like smaller enterprises applying evolutionary architecture and design principles. Likewise, smaller teams should stick to their knitting and keep their complexity in their pockets.

Further Reading

- Babar, M. A., Brown, A. W., & Mistrik, I. (2014). *Agile software architecture*: *Aligning agile processes and software architectures*. Waltham, MA: Elsevier.
- Bain, S. L. (2008). *Emergent design*: *The evolutionary nature of professional software development*. Boston, MA: Pearson Ed.
- Barbugli, E. (2014). *Lean B2B*: *Build products businesses want*. Charleston, SC: CreateSpace.
- Bass, L., & Klein, J. (2019). *Deployment and operations for software engineers*. Middletown, DE: Independent.
- Bass, L., Weber, I., & Zhu, L. (2015). *DevOps*: *A software architect's perspective*. Old Tappan, NJ: Pearson Education.
- Bente, S., Bombosch, U., & Langade, S. (2012). *Collaborative enterprise architecture*: *Enriching EA with lean, agile, and enterprise 2.0 practices*. Waltham, MA: Elsevier.
- Biehl, M. (2014). *OAuth 2.0*: *Getting started in web-API security*. Rotkreuz, Switzerland: API University Press.
- Biehl, M. (2015). *API architecture*. Rotkreuz, Switzerland: API University Press.
- Biehl, M. (2016). *RESTful API design*. Rotkreuz, Switzerland: API University Press.
- Blank, S. (2013). *Why the lean start-up changes everything*. Boston, MA: HBR Press.
- Bloomberg, J. (2013). *The agile architecture revolution*: *How cloud computing, rest-based SOA, and mobile computing are changing enterprise IT*. Hoboken, NJ: John Wiley & Sons.
- Brown, W. J., Malveau, R. C., McCormick, H. W., & Mowbray, T. J. (1998). *Antipatterns*: *Refactoring software, architectures, and projects in crisis*. New York, NY: John Wiley & Sons.
- Bruce, M., & Pereira, P. A. (2019). *Microservices in action*. Shelter Island, NY: Manning Publications.
- Carnell, J. (2017). *Spring microservices in action*. Shelter Island, NY: Manning Publications.
- Christudas, B. (2019). *Practical microservices architectural patterns*: *Event-based java microservices with spring boot and spring cloud*. New York, NY: Apress Media.
- Coplien, J. O., & Bjornvig, G. (2010). *Lean architecture for agile software development*: *Building software as if people mattered*. West Sussex, UK: 2010.
- Douglass, B. P. (2016). *Agile systems engineering*. Waltham, MA: Elsevier.
- Erder, M., & Pureur, P. (2016). *Continuous architecture*: *Sustainable architecture in an agile and cloud-centric world*. Waltham, MA: Elsevier.
- Evans, E. (2004). *Domain-driven design*: *Tackling complexity in the heart of software*. Boston, MA: Pearson Education.
- Evans, E. (2014). *Domain-driven design reference*: *Definitions and pattern summaries*. Indianapolis, IN: Dog Ear Publishing.
- Fairbanks, G. (2010). *Just enough software architecture*: *A risk-driven approach*. Boulder, CO: Marshall & Brainerd.
- Finnigan, K. (2019). *Enterprise java microservices*. Shelter Island, NY: Manning Publications.
- Ford, N. (2009). *Evolutionary architecture and design*: *19 article series*. Retrieved September 6, 2012, from

http://www.ibm.com/developerworks/java/library/j-eaed1/index.html
- Ford, N. (2012). *Emergent design and evolutionary architecture*. Retrieved September 6, 2012, from http://youtu.be/v2J2A5I5EgI
- Ford, N. (2016). *Evolutionary software architectures*. Retrieved July 7, 2016, from http://youtu.be/SzSZpZI02Jg
- Ford, N., Parsons, R. & Kua, P. (2017). *Building evolutionary architectures*: *Support constant change*. Sebastopol, CA: O'Reilly.
- Fowler, M. (1999). *Refactoring*: *Improving the design of existing code*. Boston, MA. Addison-Wesley.
- Fowler, S. J. (2017). *Production-ready microservices*: *Building standardized systems across an engineering organization*. Sebastopol, CA: O'Reilly Media.
- Garofolo, E. (2019). *Practical microservices*. *Build event-driven architectures with event sourcing and cqrs*. Raleigh, NC: Pragmatic Bookshelf
- Gilbert, J. (2018). *Cloud native development patterns and best practices*: *Practical architectural patterns for building modern, distributed cloud-native systems*. Birmingham, UK: Packt Publishing.
- Hall, G. M. (2017). *Adaptive code*: *Agile coding with design patterns and solid principles*. Redmond, WA: Microsoft Press.
- Harvick, R. (2012). *Agile architecture for service-oriented component driven enterprises*: *Encouraging rapid application development using agile*. Memphis, TN: DataThunder.
- Indrasiri, K., & Siriwardena, P. (2018). *Microservices for the enterprise*: *Designing, developing, and deploying*. New York, NY: Apress Media.
- Ingeno, J. (2018). *Software architect's handbook*: *Become a successful software architect by implementing effective architecture concepts*. Birmingham, UK: Packt Publishing.
- Kerievsky, J. (2004). *Refactoring to patterns*. Boston, MA: Addison-Wesley.
- Kersten, M. (2018). *Project to product*: *How to survive and thrive in the age of digital disruption with the flow framework*. Portland, OR: IT Revolution.
- Knapp, J. (2016). *Google venture design sprints in 90 seconds*. Retrieved April 25, 2019, from http://youtu.be/K2vSQPh6MCE
- Knapp, J. (2016). *Sprint*: *Solve big problems and test new ideas in just five days*. New York, NY: Simon & Schuster.
- Knapp, J. (2017). *Google venture design sprints*. Retrieved April 25, 2019, from http://vimeo.com/233941079
- Kocher, P. S. (2018). *Microservices and containers*. Boston, MA: Addison-Wesley.
- Krause, L. (2014). *Microservices*: *Patterns and applications*. Paris, France: Lucas Krause.
- Liebel, O. (2019). *Scalable container infrastructures with docker, kubernetes, and openshift*: *The compendium on container clusters for enterprise administrators and devops teams*. Seattle, WA: Kindle Direct Publishing.
- Millett, S., & Tune, N. (2015). *Patterns, principles, and practices of domain-driven design*. Indianapolis, IN: John Wiley & Sons.
- Nadareishvili, I., Mitra, R., McLarty, M., & Admunsen, M. (2016). *Microservice architecture*: *Aligning principles, practices, and culture*. Sebastopol, CA: O'Reilly Media.
- Newman, S. (2015). *Building microservices*. *Designing fine-grained systems*. Sebastopol, CA: O'Reilly Media.
- Newman, S. (2019). *Monolith to microservices*: *Evolutionary patterns to transform your monolith*. Sebastopol, CA: O'Reilly.
- Pacheco, V. F. (2018). *Microservice patterns and best practices*: *Explore patterns like cqrs and event sourcing to create scalable, maintainable, and testable microservices*. Birmingham, UK: Packt Publishing.
- Ratcliff, L., & McNeill, M. (2012). *Agile experience design*: *A digital designer's guide to agile, lean, and continuous*. Berkeley, CA: New Riders.
- Richards, M., & Ford, N. (2020). *Fundamentals of software architecture*: *A comprehensive guide to patterns, characteristics, and best practices*. Sebastopol, CA: O'Reilly Media.
- Richardson, C. (2019). *Microservices patterns*: *With examples in java*. Shelter Island, NY: Manning Publications.
- Rico, D. F. (2012). *Architecture and design practices for agile project management*. Fairfax, VA: Gantthead.Com.
- Rico, D. F. (2012). *Principles of evolutionary architecture and design for agile project management*. Fairfax, VA: Gantthead.
- Rico, D. F. (2014). *18 reasons why agile cost of quality* (*CoQ*) *is a fraction of traditional methods*. Retrieved June 25, 2014, from http://davidfrico.com/agile-vs-trad-coq.pdf
- Rico, D. F. (2017). *U.S. DoD vs. Amazon*: *18 architectural principles to build fighter jets like amazon web services using devops*. Retrieved January 26, 2017, from http://davidfrico.com/dod-agile-principles.pdf
- Rico, D. F. (2018). *Lean & agile contracts*: *21 principles of collaborative contracts and relationships*. Retrieved June 29, 2018, from http://davidfrico.com/collaborative-contract-principles.pdf
- Rico, D. F. (2019). *Business value of CI, CD, & DevOpsSec*: *Scaling up to billion user global systems of systems using end-to-end automation and containerized docker ubuntu images*. Retrieved August 23, 2019, from http://davidfrico.com/rico19c.pdf
- Rico, D. F. (2019). *Lean leadership*: *22 attributes of successful business executives, directors, and managers*. Retrieved August 28, 2019, from http://davidfrico.com/lean-leadership-principles.pdf
- Rodger, R. (2018). *The tao of microservices*. Shelter Island, NY: Manning Publications.
- Sharma, S. (2017). *Mastering microservices with java 9*: *Build domain-driven microservice-based applications with spring, spring cloud, and angular*. Birmingham, UK: Packt Publishing.
- Shore, J. (2011). *Evolutionary design illustrated*. Retrieved September 6, 2012, from http://vimeo.com/151616935
- Suryanarayana, G., Smarthyam, G., & Sharma, T. (2015). *Refactoring for software design smells*: *Managing technical debt*. Waltham, MA: Elsevier.
- Van Drongelen, M., & Dennis, A. (2017). *Lean mobile app development*: *Apply lean startup methodologies to develop successful ios and android apps*. Birmingham, UK: Packt Publishing.
- Vernon, V. (2013). *Implementing domain-driven design*. Upper Saddle River, NJ: Pearson Education.
- Vernon, V. (2016). *Domain-driven design distilled*. Upper Saddle River, NJ: Pearson Education.

CASE STUDIES OF SUCCESSFUL EVOLUTIONARY AND EMERGENT ARCHITECTURE AND DESIGN PRINCIPLES

- Mainframe Apps. *The global industry for commercial software applications is one of the best examples of evolutionary architecture. Although crude mechanical calculators existed for millennia and the difference engine was perfected in the early 1800s, the first electronic computer wasn't created until 1945. As a result of World War II, the ENIAC was designed to make automatic ballistics calculations for the U.S. military, a discipline held by human mathematicians or computers who performed their calculations underneath the decks of battleships as conflicts ensued. After World War II the next major user was the business world itself. The first mainframe computer, UNIVAC, was designed in 1951, and IBM created the first mass marketed mainframe by 1964, IBM 360. Early mainframe producers were known as IBM and the Seven Dwarfs, including DEC, CDC, GE, RCA, Univac, Burroughs, and Honeywell. These early mainframes came without middleware or applications and software was produced by purveyors of custom software services. Early operating systems took thousands of programmers, decades and billions of dollars to create. It was these early experiences that inspired IBM managers to document basic software project management and software engineering principles. With the plethora of mainframes, this opened the door for third-party reusable commercial software applications. Programmers created custom applications for one client and commercialized them for the mass market. This formula hasn't changed much today. IBM was not pleased with this practice and sued purveyors of third-party IBM software applications. The U.S. Justice Department threatened to break IBM in two, one part for hardware and one part for software in order to prevent a market monopoly for mainframe products and services. With legal division in-mind, IBM stopped suing application developers, opening the door for the U.S. and global commercial application software industry. Since software is infinitely more malleable than hardware, mainframe applications grew from a few dozen and hundred in the 1960s and early 1970s to literally thousands of software applications by 1980. Computer hardware itself became better, faster, and cheaper with the advent of midrange computers, minicomputers, engineering workstations, and the personal computer itself. Global software industry revenues soon eclipsed hardware, because of the ability to rapidly release newer and better varieties of software applications with increasing frequency. Although IBM was an early purveyor of linear long-term software management and engineering practices designed for decade long product cycles, this was soon to change. The software industry pioneered rapid prototyping, iterative, incremental, spiral, and evolutionary development, joint, rapid, and participatory application design, and modern lean and agile methods themselves. The software industry grew from a free service offered to mainframe customers in the 1960s to more than $300 billion annually today and $5 trillion for the IT industry in total.*
- Internet Apps. *The world-wide-web or WWW is an excellent example of bottoms-up, evolutionary, and emergent architecture and design principles. Although the Internet itself had its earliest beginnings in the 1960s and 1970s, it didn't really skyrocket or take off until the late 1980s and early 1990s, when Tim Berners-Lee is credited with inventing the Hyper Text Transfer Protocol (HTTP), Web Browser and WWW itself. Prior to that, government, industry, and academic researchers were busy designing and implementing the fundamental building blocks of the WWW in the form of TCP Internet Protocols, File Transfer Protocols, E-Mail programs, and basic data and image formats such as GIF, JPEG, and TIF. Computer and software engineering firms such as IBM, Intel, and Microsoft among others were busy commoditizing Personal Computers (PCs) with the power of mainframes and super computers that were once the exclusive domain of large military organizations, government and aerospace agencies, and blue-chip Fortune 500 corporate conglomerates with massive amounts of resources, hierarchies, and centralized resources. Even the creators of early commercial Bulletin Board Systems (BBSs) such as America Online (AoL), Prodigy, and CompuServe chipped into to popularize the use of Modems and create a global commercial market for online Internet collaboration. With the combination and synergy of the so-called WinTel platform, Internet, basic WWW building blocks, and early adopters of BBBs, people like Marc Andreessen created the first mass produced commercial Web Browsers and the WWW itself exploded onto the global scene with a few million end users by the early 1990s. Even the founder of Intel, who commercialized transistors, memories, and microprocessors, chipped in the ever popular (Gordon) Moore's Law, which stated "Microprocessors halved in size and doubled in speed every 18 months!" With the WWW building blocks and Moore's Law in hand, these early WWW product and service designers began evolving the WWW components in a bottom up, evolutionary, and emergent manner, not only exponentially amalgamating more and more raw numbers of sophisticated innovatively new building blocks but increasing their capacity, efficiency, and doubling their speed in short intervals, often distributing their wares at little to no cost. End users themselves exponentially flocked to the WWW in droves, tens of millions at a time, to the present era where there over 4 billion end users of the WWW itself and nearly one trillion Internet hosts or IoT devices connected to the Internet. The most amazing part of this case study is that the Internet and WWW grew in a highly organic, evolutionary, and emergent style like biological organisms in a primordial soup without a top-down Big Design Up Front (BDUF), Enterprise Architecture (EA), or Systems of Systems (SoS) blueprint. The WWW continues to evolve, increase in performance, scale, utility, user experience, and security, and is the fundamental building block for economic, political, and cultural globalization.*
- Smartphone Apps. *Smartphones are another excellent example of evolutionary design principles. Although telephones were created in 1875 and reached the height of popularity by 1925, commercial portable wireless cell phone weren't created until 1973. By then, there were millions global end users that were addicted to talking on telephones. There was a ready-made market and telephones came in every shape, size, color, price, performance category, and variety. The market for cell phones grew slowly from 1970 to 1990, as they were heavy, bulky, expensive, unreliable, had limited range and battery life, few wireless repeaters existed, and there were few end users. Like the WWW itself, early smartphones took advantage of a sudden synergy and availability of low-cost and high-performance commercialized computer components. These included microprocessors, memories, flash storage, touchscreens, operating systems, networking protocols, and the Internet. Even firms like Microsoft attempted to slim down their computer operating systems to offer point-and-click Windows-based smartphone operating systems. Although Motorola was one of the first mass marketers of early cellphones, firms like IBM are credited with creating one of the first smartphones—Simon and the Canadian firm Blackberry stoked the demand for the proliferation of business market facing smart phones. Later on, the Scandinavian firm Nokia dominated the market for early smartphones and then Apple entered the market and dominated the smartphone market for nearly two decades. By 2007, Android entered the market and capitalized upon the well-established customer base for lower-priced, better featured smartphones. Today, Android apps outnumber Apple iPhone apps by a ratio of 3:2. Although the smartphone hardware platform is an exemplar of evolutionary architecture itself, exhibiting Moore's Law by halving in price and doubling in speed every 18 months or less, the software ecosystem that operates on top of the Android OS is an even better example of emergent design. With only a few hundred Android apps in 2007, Android apps eclipsed the smartphone app market in only 10 years with nearly 3 million of them. Many of them exist as two-pizza team fully containerized microservices or as even smaller single-function widgets that provide fast and extremely convenient features, information, and data. Smartphone apps based on emerging microservices architecture principles literally evolve at the speed of light and can be deployed to billions of smartphones throughout the globe hundreds and even thousands of times per week for each app if necessary. Smart phone apps embody most lean thinking and lean startup values, principles, and practices and designers tease out tacit market knowledge to optimize user experience and business value of their wares with expert ease. Although microservices are a great alternative to bloated PC apps, widget architectures are an even better alternative to bloated microservices apps for wearable devices like smartwatches—The next market disruption!*