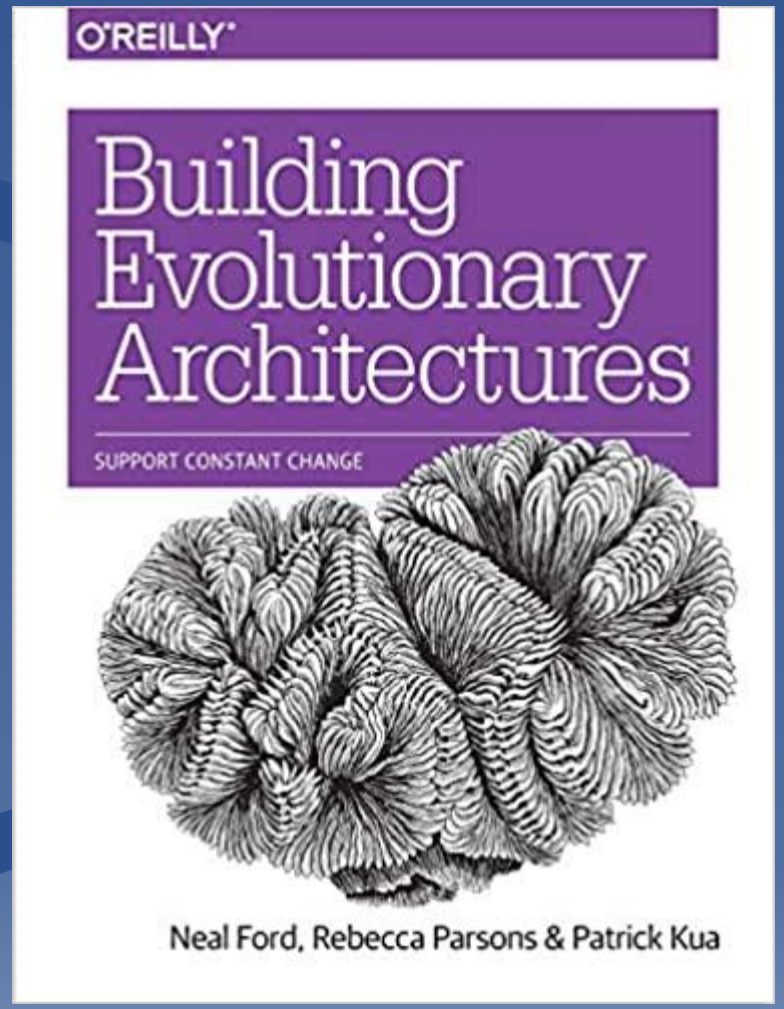
Evolutionary Architectures

D. Vanderbist (15/7/2021)



Introduction

Software Architecture?

View was challenged by Agile methodology

Old View:

- Architecture equivalent for software from construction industry:
  - plan first (blueprint) build later
- Successful architecture was something that didn't need to change during development:
  - Avoid to scrap and rework
- Requirement before coding leads to phased or waterfall approach
  - Fixed requirement

New View:

- Regular changes to the requirements have become part of normal business Evolution
- From architecture as an up-front challenge to a continuous challenge

This is called Evolutionary Architecture:

- Changes keep coming so development never ends and the so architecture will keep changing
- Compare to a hotel that has continuous changes and refurbishments whilst still serving guests

Fitness Function concept to monitor the state of architecture.

Software Architecture?

"Software Architecture":
- Defined as "the important stuff"
- "Balancing the important things"

Architects:
- Analyze business and domain requirements
- Find a solution to balance all concerns optimally
- Balancing will cause inevitable clashes between competing factors

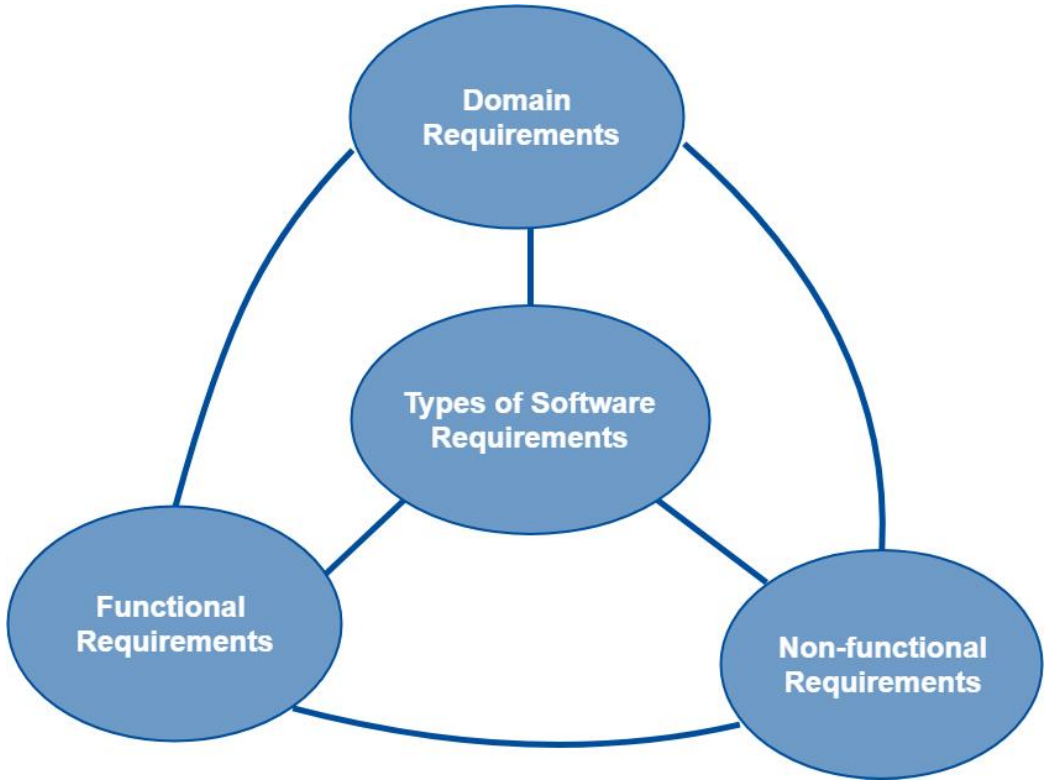
Architectural concerns:
- Technical requirements
- The *-abilities

- The *-abilities:
  - Accessibility
  - Accountability
  - Accuracy
  - Adaptability
  - Administrability
  - Affordability
  - Agility
  - Auditability
  - Autonomy
  - Availability
  - Compatibility
  - Compossibility
  - Configurability
  - Correctness
  - Credibility
  - Customizability
  - Debuggability
  - Degradability
  - Demonstrability
  - Dependability
  - Deployability
  - Determinability
  - Discoverability
  - Distributability
  - Durability
  - Effectiveness
  - Efficiency
  - Extensibility
  - Failure transparency
  - Fault tolerance
  - Fidelity
  - Flexibility
  - Inspectability
  - Installability
  - Integrity
  - Interoperability
  - Learnability
  - Maintainability
  - Manageability
  - Mobility
  - Modifiability
  - Modularity
  - Operability
  - Orthogonality
  - Portability
  - Precision
  - Predictability
  - Process capabilities
  - Producibility
  - Provability
  - Recoverability
  - Relevance
  - Reliability
  - Repeatability
  - Resilience
  - Responsiveness
  - Reusability
  - Robustness
  - Safety
  - Scalability
  - Seamlessness
  - Security
  - Self-sustainability
  - Serviceability
  - Simplicity
  - Stability
  - Standards compliance
  - Survivability
  - Sustainability
  - Tailorability
  - Testability
  - Timeliness
  - Traceability
  - Usability

Software Architecture?

Types of requirements

- Functional requirements aka Business Requirements
- Non-functional requirements aka Technical Requirements
- Domain requirements aka Industry Standard Requirements



Software Architecture?

Why taking change into account?

- Systems become harder to modify over time: Bit Rot or the gradually degrade/decay over time as architects have chosen architectural patterns to implement the required abilities but the characteristics can decay over time
- Changes are triggered by changing functionality or scope of systems
- Changes occurs outside the architect's long term plan or vision

How do long term planning if things are changing all the time?

- Change is inevitable:
  - We can't predict the change but we know it is coming
  - So we build change into the architecture
- What is the alternative for fixed plan?
  - Reason for a fixed plan is financial as changes are costly
  - DevOps is a solution for this
- For example layers in a system help to isolate change but often Developers want to bypass theses layers

Evolutionary architecture adds
a time element and changes into architecture

Evolutionary is a meta-characteristic
a wrapper characteristic to protect
all other characteristics

Continuous Architecture

Continuous Architecture

- Architecture that has no end-state as everything keeps evolving
- Supports guided, incremental change across multiple dimensions
  - Incremental: changes are gradually released to a (subset) of clients supported by CI/CD and DevOps techniques
  - Guided : to protect architectural characteristics through the use of fitness functions

Change over Time

Guided Changes towards end-state

Fitness functions:

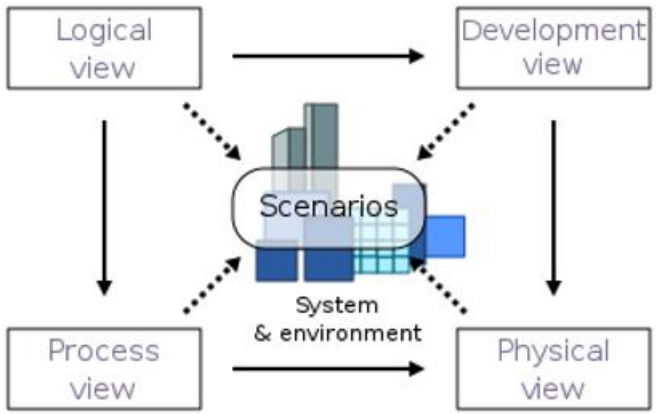
- Summarizes how close a prospective design a solution is to achieving the set aims
- Stems from evolutionary computing: How has the algorithm improved over time?
- Same applies to architecture: How have changes impacted the important characteristics of the architecture to prevent degradation over time?
- Turns an unconstrained, irresponsible approach into an approach that balances need for rapid changes with rigor around architectural characteristics
- Allows for comparison and debate: the general state of the architecture relative to the fitness function

For evolutionary architecture to work architects have to think beyond the technical architecture dimensions.

Architectural Viewpoints

- Technical: components and implementation
- Data: data schemas
- Security: policies and guidelines
- Operational/Systems: logical to physical mapping

Example: 4+1 Architecture View Model



- **Logical view:** The logical view is concerned with the functionality that the system provides to end-users.
- **Process view:** The process view deals with the dynamic aspects of the system, explains the system processes and how they communicate, and focuses on the run time behavior of the system.
- **Development view:** The development view illustrates a system from a programmer's perspective and is concerned with software management. This view is also known as the implementation view
- **Physical view:** The physical view depicts the system from a system engineer's point of view. It is concerned with the topology of software components on the physical layer as well as the physical connections between these components.
- **Scenarios:** The description of an architecture is illustrated using a small set of use cases, or scenarios, which become a fifth view. The scenarios describe sequences of interactions between objects and between processes.

Fitness Functions

What?

- An architectural fitness function provides an objective integrity assessment of some architectural characteristic
  - Embodies the protection of *-abilities of a system
- System wide fitness functions allow to do trad-offs across multiple dimensions.
  - Difficulties is creating an unified way to compare the result of different fitness functions: quantification issues
  - A system is never the sum of its parts it is the product of the interactions of its parts.

Fitness Functions Examples:

- More obvious:
  - Performance test: performance met or not
  - Code standard adherence: cyclomatic complexity
- Less obvious
  - Failover: how to test a successful failover mechanism?

Fitness Functions

Categories of fitness functions:

- Atomic vs Holistic:
  - Atomic run against a singular context and focus on one particular aspect of the architecture
  - Holistic run against a shared context and focus on a combination of architectural aspects
- Triggered vs Continual:
  - Execution cadence of the fitness functions
  - Triggered: based on an event (e.g. unit test during builds)
  - Continual: not linked to a schedule and constantly verified
  - Monitoring-Driven Development (MDD), using monitors in production to asses technical and business health
- Static vs Dynamic:
  - Static: distinct, fixed results like pass/fail
  - Dynamic: range of values related to a context
- Automatic vs Manual:
  - Automatic: in a CI/CD and DevOps context
  - Manual: outside an automated context
- Temporal:
  - Run triggered by change
  - Run on a specific occasion
- Intentional vs Emergent
  - Intentional: defined at the beginning/inception
  - Emergent: become apparent during development

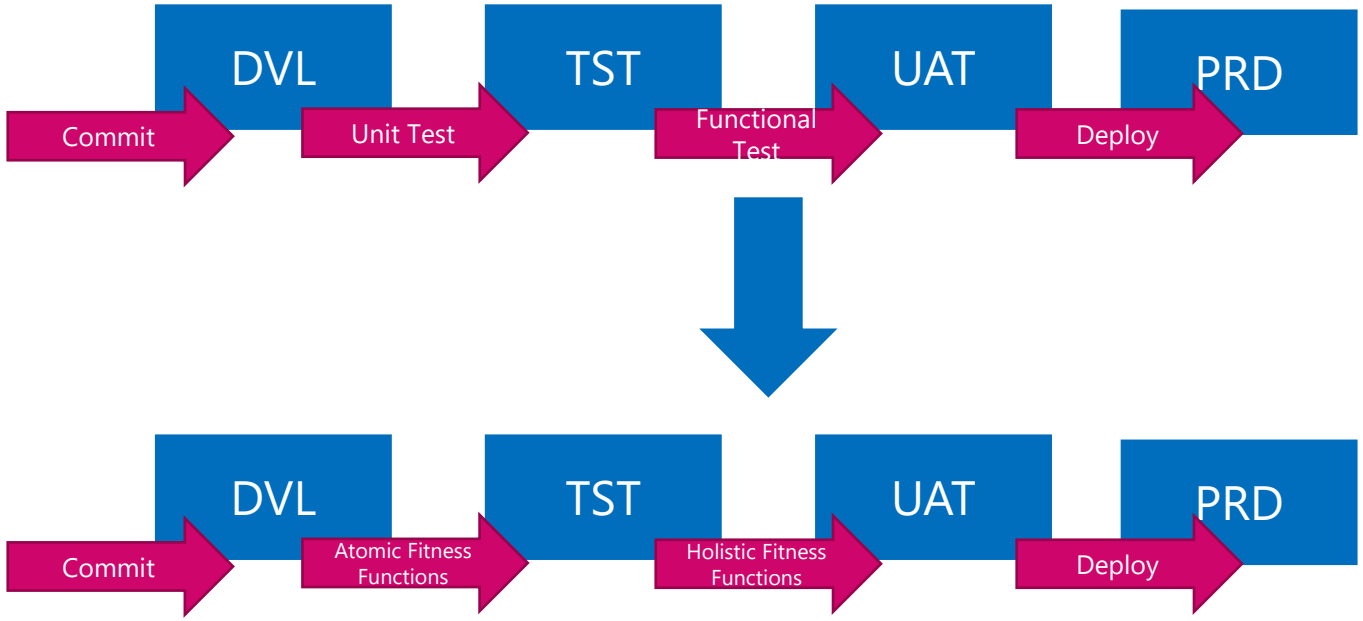
Incremental Change

Changing the CI/CD pipeline by adding fitness functions

- Atomic & Triggered: part of unit and functional test build pipeline
- Holistic & Triggered: integration test of deployment pipeline
- Atomic & Continual: continuous end-point testing e.g. rest calls are using correct verbs and error handling
- Holistic & Continual: test fault tolerance and robustness e.g. chaos monkey to test production behavior

Changing requirements gathering:

- Hypothesis Driven Development: no formal requirements gathering but building hypotheses during ideation instead of gather formal requirements



Architectural Coupling

Decoupling:

- Appropriate coupling:
  - The right level of architectural coupling
  - Find the right balance between maximum benefit of coupling with minimal overhead and costs
- Logical coupling:
  - Module modularity
  - Memory linked libraries
- Physical coupling:
  - Packing of modules
  - Network linked components

Quantum:

- Minimum amount of any physical entity involved in an interaction
- Architectural quantum: independently deployable component with high functional cohesion
- Architects determine their quantum size:
  - Small quanta implies faster change because of smaller scope
  - The lower bound of incremental change
- Natural Quantum:
  - Component are very hard to break down in smaller units
  - Transactional context is another hard to break force in an architecture

Architectural Styles:

- Monolithic architecture:
  - Quantum is the full system as everything is tightly coupled
- Microservices architecture
  - Quantum are the physical bounded contexts
- Architecture style influence the architectural quantum
  - Possibilities for Incremental changes
  - Possibilities for guided changes with fitness function
  - Appropriate level of coupling

Evolutionary Architectures

Architecture: Big Ball of Mud

Big Ball of Mud

What?

- Extremely low score on evolutionary because everything is tightly coupled it is just a collection of classes

Evolutionary Fitness:

- Incremental changes: doesn't allow for incremental change
- Guided change: no guided change with fitness functions, no compartments
- Appropriate Coupling: no appropriate coupling

Architecture: Monoliths

Unstructured Monoliths

What?

- Unstructured monoliths are a collection of loosely connected classes
- UI on interconnected classes

Evolutionary Fitness:

- Incremental change: Large quantum sizes hinder incremental change
- Guided change: existing tools for testing that can be used in fitness functions. But typically these architecture do not scale so making tests for performance and scaling is hard
- Appropriate coupling: limited to no internal structure beside the connections between classes

Layered Monoliths

What?

- Are a logical distribution not a physical
- E.g. presentation, business rules, persistence, DB
- Layers are responsible for decoupling: isolation of concerns with interfaces between layers

Evolutionary Fitness

- Incremental change:
  - ok if change limited to a layer
  - cross layers changes are more difficult if they are also the work organization in teams: UI teams needs to talk to team responsible for business logic
- Guided change: isolation allows to test more part separately via fitness functions
- Appropriate coupling: architecture is easy to understand because of the layers

Architecture: Monoliths

Modular Monoliths

What?

- It is the microservices equivalent for monoliths.
- If decoupling in modules is done correct the same benefits as for microservices can be obtained with modules in a monolith architecture.

Evolutionary Fitness:

- Incremental change: Incremental changes to functionality isolated in modules: one module is one logical grouped functionality
- Guided change: mocking and testing work better if isolation is more fine granular
- Appropriate coupling: low coupling achieved through functional isolation

Microkernels

- Is also a monolithic architecture style
  - Core system with API based plugged-in extensions (hook-ins or extension points)
  - Isolation of the extensions from the core system
- Architecture works fine as long as the extension points do not need to coordinate between each other to get the work done:
  - If coordination is required information is exchanged via the core because plugins should not communicate directly
  - Version management of the plug-in becomes important reducing the stateless level of the system
  - Versions are semantic couplings between components

Evolutionary Fitness:

- Incremental change: if they are limited to the plugins and the core system is mature and stable
- Guided change: split between the core and plug-ins to measure fitness
- Appropriate coupling: checking the dependencies between plug-ins is key to check for the right level of coupling

Architecture: Event Driven Architectures (EDA)

Event Driven Architectures (EDA)

- Combines disparate systems via message queues
- Allows for parallel processing
- Two types:
  - Brokers
  - Mediators

Brokers

What?

- Building Blocks:
  - Message queues
  - Initiating events
  - Intra-process events
  - Event processors
- Typical design challenges are robust error handling since there is no central component
- Architecture is extremely evolutive

Evolutionary Fitness:

- Incremental change: Loosely coupled services allow for non-breaking changes, small and self-contained services
- Guided change: atomic fitness functions are easy but Holistic fitness functions are complex, solved via correlation ID's to track cross-service behavior
- Appropriate coupling: low degree of coupling

Architecture: Event Driven Architectures (EDA)

Mediator

What?

- Process manager that manages queues and moves messages to the correct processor
- Mediator handles the coordination, processors do not call each other
- Transactional coordination is an advantage: all processing steps must be done before mediator gives final ok to caller, single end status

Evolutionary Fitness:

- Incremental change: small and self-contained services
- Guided change: fitness for processes is easier to check than for mediator
- Appropriate coupling: the mediator contains domain logic increasing the coupling in this component

Architecture: Service-Oriented Architectures (SOA)

Service-Oriented Architectures (SOA)

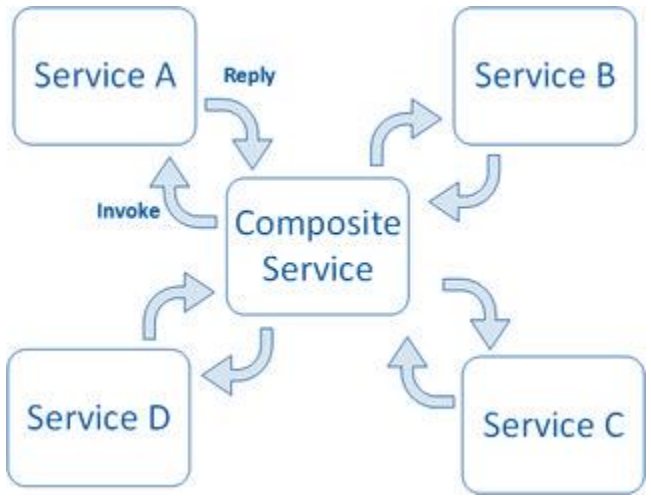
- Types
  - ESB-Driven
  - Micro-Services
  - Service Based

ESB-Driven

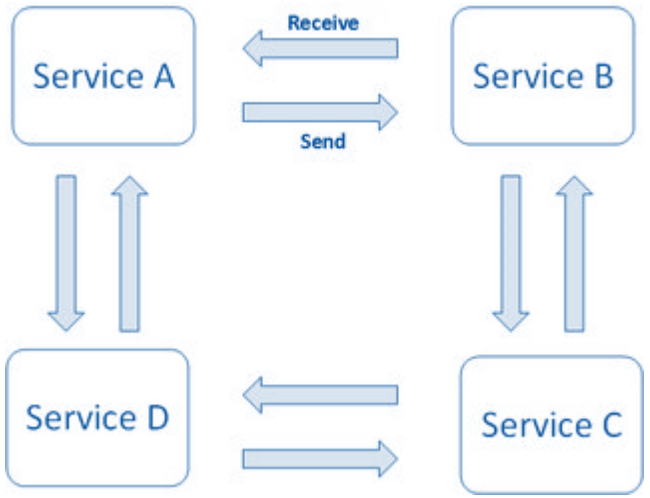
What?

- Service bus act as mediator and handles typical integration plumbing: transformation, choreography
- Same building blocks like EDA but the organization of the services differ
- Orchestration vs. Choreography:
  - Differ where the logic that controls the interactions between the services involved should reside.

Orchestration



Choreography



Architecture: Service-Oriented Architectures (SOA)

ESB-Driven

What?

- Layers:
    - Business Services: course grained BPEL => abstract, high-level
        - Message Bus: stitching things together
        - Process Choreographer
        - Service Orchestrator
    - Enterprise Services: concrete, implemented, detailed services
    - Applications Services & Infrastructure Services
- Responsibilities
    - Mediation & Routing: locate and communicate with services
    - Choreography & orchestration: combines enterprise services for business service purposes
    - Message enhancement & transformation: protocol and message transformations
- ESB massive architectural quantum:
    - Equal to monolith but distributed: makes singular evolutionary changes very hard
    - Compared to Microservices: bound context, every services has its entities end if they overlap no effort is put in to align these objects

Evolutionary Fitness:

- Incremental change: services allows for reuse and segregation of resource but is not easy to maintain and have isolated changes
- Guided change: hard to test as every piece is part of a workflow
- Appropriate coupling: If components are reusable it make sense, but in general SOA architecture are not built for incremental growth

Architecture: Service-Oriented Architectures (SOA)

Microservices

What?

- Modelled around business domains:
- Combines Continuous Delivery concepts and logical partitioning through the bound context concept.
  - Highly decentralized and failure isolation
  - Highly observable monitorable
- The share nothing architecture
  - Separate business, persistence and DB layer for a set of entities into a service accessed via a common UI
  - Each service is defined around Domain Driven Design (DDD)
  - Communication is organized using messaging: REST or queues
  - Interface and boundaries hide technical implementation
- Why not adapted earlier:
  - CI/CD practices and containers were not available some time ago

Evolutionary Fitness:

- Incremental change: bounded context makes it easy to limit changes to the context, layered services isolate changes in the services to one layer
- Guided change: allow for atomic and holistic fitness functions
- Appropriate coupling:
  - Two types of coupling integration and services.
  - Services calling each other to pass information: share nothing means in principle share only the appropriate to have a minimal level of coupling

Service-Based Architectures

What?

- A variant on micro-services architecture, but:
  - Large service granularity: course grained services
  - DB scope: one overall DB instead of a DB per service
  - Integration middleware: coordination between services externalized to service bus or mediator

Evolutionary Fitness:

- Incremental change: incremental change is more functional according to the domain split of services
- Guided change: more difficult than microservices because of increased coupling and larger bound contexts
- Appropriate coupling: domain centric coupling can be an appropriate level leaning on DB schema

Architecture: Serverless Architectures

Serverless

What?

- Backend-as-a-Service (BaaS)
- Function-as-a-Service (FaaS)

Evolutionary Fitness:

- Incremental change: incremental change is only about deploying code as the infrastructure is managed
- Guided change: fitness is ensuring integration points keep working
- Appropriate coupling: removes a lot of concerns: technical, operations, security are organized as part of the service

Evolutionary Data

DB Schema's

Evolving DB schemas requires changes to be:

- Tested: test ORM mappers after changes
- Versioned: source code and DB schema are connected
- Incremental: controllable set of changes

Schema Architecture and Operations:

- Using shared DB between multiple applications
    - Changes for one of these might break the other applications
- Replace Schema Change Undo with Compensating Actions
    - Often undo for DB schemas changes is not provided anymore because complexity
    - Undo:
        - Allows for forward and backward compatibility, but has practical implications (e.g. what about data stored in new columns and undo is executed or a drop table)
        - Undo is replaced by compensating action: instead of undo we do an additional DB migration removing the column

Solution is DB migration in 3 phases:

- Start
- Transition: expand schema, having the old and the new in parallel
- End: contract schema, removing old storage structures

Evolving Schema Example:

- Split stored consumers names into first and last name columns
- To allow application to use the old and the new structure during the time application are migration from the old to the new structure we need:
    - triggers on insert or change to update the new columns based on a functions that splits and stores the old
      and/or
      triggers on the update of the new columns to store the information in the old combined column
- Prepare without side effect:
    - Add columns that are allowed to be null
    - Make sure we never use a "*" to CRUD but always use a fix list of columns

Impact of DB Transactions

- Coupling because of transactional context is hard to track
- Limit transactional context limits the level of coupling
- Some architecture are better or worse for heavy transactional context worse e.g. microservices

Drivers for change:

- DB Schema's change less frequently than applications:
    - Often tables are added in order to avoid DB schema changes of existing tables
    - Adding and not removing degrades the usefulness of the domain abstraction over time
- Data quality is important:
    - Data not used by any application anymore should be removed
    - Requires overview of which application uses what
    - There might be value in historic data but it also limits the possibility of changes later

Evolutionary Architectures in Practice

Building Evolutionary Architectures

Introducing Fitness Functions:

- Identify: Identify the important dimensions affected by evolution
- Define: Define fitness functions for each dimension
- Execute: Automate, embed and execute fitness functions as part of CI/CD approach. to encourage incremental changes

Fitness functions require testability so affect architecture: testability and layering are to be balanced and traded-offs to be made

Embedding Evolutionary Architecture:

- Greenfield:
  - Easier as no technical debt or limitations because of existing implementation
  - No need to untangle old structure and architectures
  - Add aspects to CI/CD pipeline in the right way instead of having to retro-fit them
- Retrofitting:
  - Appropriate coupling and cohesion: technical cohesion for components and DB makes the work easy, functional cohesion determines the granularity of components
  - Engineering practices: industrialization starts for the sake of efficiency but can become a driver for more high-level things like evolutionary architectures
  - Typical impediment is operations: evolutionary architecture requires easy deployment of incremental changes
  - Types of retrofitting:
    - Refactoring: changing the structure of code without change external behavior change
    - Restructuring: changing structure and behavior of code
- Commercial of the shelve (COTS):
  - Incremental change: very difficult as extensions can be tested but often the system itself is a black box
  - Fitness functions: often impossible as systems do not expose enough internals
  - Appropriate coupling: API often are ok but not for the last 10% which makes it not just enough to integrate
  - DB schema and storage is often black box

Building Evolutionary Architectures

Migrating Architecture

- Risk:
  - People like to write a framework rather than using it
  - Prefer self-build over reuse or open source
  - Meta-work is more interesting than real work
- Consider:
  - Service granularity
  - Transactional boundaries
  - DB issues (read schema, shared DB or not)
  - Shared framework and libraries (version control)
- Advise
  - Avoid building systems that mimic the business communication, organizational structure
  - Break transactional boundaries were possible
  - Achieve small deployable units to allow for incremental change i.e. a fine grained release schedule

Migrating shared modules:

- Clean code split
- Clean split of functionality
- Code sharing but separate deployment
- Duplicating the module

Evolution Guidelines:

- Remove needless variety
  - Immutable infrastructure: programmatically defined infrastructure or infrastructure as code, controlling the environment
- Make decisions reversible:
  - Blue/green deployments: swapping staging and production environments
  - Feature toggles
- Prefer evolvable over predictable:
  - If we can deal easily with changes we don't need to know everything upfront
- Build anticorruption layers:
  - Don't write code for a specific technology: functionality too close to a used library, wrap a library in more generic functions describing the what instead of the how
  - Service templates: discovery, monitoring, logging, metrics, authentication, authorization all part of a template
- Build sacrificial architectures:
  - A POC to throw away if successful
  - Cloud makes sacrificial architecture more attractive as not a lot is invested in infrastructure
  - Risk of not throwing away functionality leads to adding more and more

Building Evolutionary Architectures

Evolution Guidelines:

- Mitigate External Change
  - External dependencies on tools, libraries updated via the internet
  - Version management of external repositories: pull model for changes instead of a push model
  - Set-up an internal repository
- Updating Libraries vs Frameworks
  - Difference: custom code calls a library vs, a framework calls custom code or functional calls vs subclassing
  - Libraries are preferred because to introduce less coupling
- Prefer Continuous Delivery over Snapshots
  - Often snapshots are used out of fear of testing with continuous updates
  - Snapshots are from an era where we wanted to isolate changes as much as possible whereas evolutionary architecture expect all things to change all the time kept under control through fitness functions
  - Changes are fluid and guarded
- Version Service Internally:
  - Service endpoint versions
  - Through version numbering or through internal resolution
  - Internal resolution return the correct version based on build in logic

Evolutionary Architectures: Pitfalls and Antipatterns

- Pitfalls (PF): bad all the way, all the time
- Antipattern (AP): pattern that looks initially good but turns out the be bad, most of the time better alternative exist

Technical Architecture::

Vendor King (AP)

- Large ERP tools work when a company is willing to bend business processes to follow the tool. Works and gives benefits when architect understand the impact and limitations.
- When company is ambitious and build an eco-system around the tool it basically couples the organization to the tool. The "Tool = core + extensions", since the core is the tool, the architects cannot avoid the core making the vendor the king of the architecture
- The solution is to treat the platform like any integration point even if it contains a lot of responsibilities of the business
- The Last 10% trap when the tool cannot fully optimize the workflow, change the workflows rather than changing the tool
- "Let's stop working and call it a success" risk: ERP require huge investments and no one want to admit the project is a failure

Leaky Abstractions (PF)

- Software is build on a tower of abstractions: OS, OSI model, frameworks
- Abstractions to avoid we have to think about the lowest levels
- Errors in lower abstraction layers will bubble up to the higher layers as they are unexpected: unhandled errors
- Always understand one layer below the one you are working in.
- Define invariants at layer interfaces through fitness functions so they are controlled,

Evolutionary Architectures: Pitfall and Antipattern

Technical Architecture:

Last 10% Trap (AP)

- The last 10% will leave the project in disappointment

Code Reuse Abuse (AP)

- "reuse is more like organ transplant than snapping together Lego blocks"
- Making something reusable is difficult
- Making something reusable might harm more than it generates benefits. Adding extension points to allow the code the behave in multiple useful ways. This make is more difficult to reuse for one specific usage. The genericity hinders the specificity,
- Microservices prefer duplication over coupling through reuse

Resume-Driven Development (PF)

- Developers want to use the latest and greatest to add as experience to their Resumé
- Focus must be on best architecture to solve the problem domain

Incremental Change

Inappropriate Governance (AP)

- Maximize shared resources as cost-saving measure problem this also increases coupling
- Costs of overengineering
- Goldilocks Governance model: pick 3 technology stacks for standardization easy, intermediate, complex
- Only allow service requirements to drive the stack selection

Lock of Speed to Release (PF)

- Strong correlation between the speed to release and the ability to evolve software design
- Measured through Cycle Time: initiation to completion of unit of work, evolutionary speed is proportional to the cycle time

Evolutionary Architectures: Pitfall and Antipattern

Business Concerns:

Product Customization (PF)

- Sales will sell any feature before thinking whether or not it can be built.
- Customizable software comes at a cost:
    - Unique build for each customer
    - Permanent feature toggles
    - Product-driven customization

Reporting (AP)

- Using same DB schema for systems of record and reporting: problem is that reporting does not need same layered architecture to support its function
- Consistency is more important than transactional behavior: reporting can be done on an event stream using a denormalized DB optimized for reporting,
    - Event systems for reporting and BI,
    - Relational systems for systems of record

Planning Horizons (PF)

- Planning drives the need for assumptions and early decision taking, but this means the decisions are taken with the least amount of information leaves no opportunity to revisit plans
- Spending time on assumptions through study, comparison: more time means typically we are more reluctant to let these assumptions go because of strong attachment, same for handcrafted artifacts
- Planning cycles should avoid triggering irreversible decisions

Evolutionary Architecture in Practice

Organizational Factors:

- Cross functional teams:
  - Eliminate operational friction, coordinate friction
  - Roles business analyses, architecture, testing, operations and data
- Organization around business capabilities, not around job functions: focus on functional aspects not on technical layers
- Product over Project:
  - Developers should be involved in the operational aspects of their solution (product) and not stop once a problem is solved (project)
  - Focus on the long life of a product give long-term company buy-in
  - Products: have a long lifespan, live forever, have owners to manage requirements
  - Projects: have a fixed duration
- Team size:
  - Amazon: team size is two pizza, teams must be able to be fed by two large pizza's
- External Change:
  - Interface contracts are consumer driven contracts
  - Provider can evolve in any way as long as the contracts are respected
- Team member connections: the more member the more connections are needed, n*(n-1)/2 for n team members.

Team coupling characteristics:

- Culture:
  - Do al team members understand the fitness functions
  - Measuring fitness will result in behavior adaption: how something is measured influences the behavior
  - Culture of experimentation: small activities on a regular basis to try out new things
- How to encourage:
  - Bring ideas from outside,
  - Encourage explicit improvement,
  - Spike a stabilize,
  - Allow innovation time,
  - Apply set based development (exploring multiple options at the same time; more competing solutions will result in one better solution),
  - Connecting engineers with end-users

CFO & Budgeting:

- Cost of architecture decreases when number of quanta increases
- Benefit decreases over time: means there is an optimal number
- Avoid best practices guides and use evolutionary architecture instead

Evolutionary Architecture in Practice

Building Enterprise Fitness Functions

- Starting points:
  - Low-hanging fruit: systems with appropriate cohesion trough decoupling
  - Prioritize according to highest value, lack of testing, solving infrastructure dysfunction (operational part)
- Future state?
  - AI driven fitness functions, generative testing (edge cases generated on statistics)

Why or Why Not Evolutionary Architectures

- Why?
  - Predictable vs Evolvable: the predictable stable will not see the market disruptions, existing companies are worse of than new just entering a market (existing have legacy systems and technical debt, new have all the flexibility)
  - Scale: eco-systems must be able to scale, traditional transactional architectures do not scale and create a DB bottleneck
  - Advance business capabilities: incremental changes allow for hypotheses and data-driven development, fast feedback cycles
  - Cycle time as business metric: time to market is a business differentiator technical enabled through fast software cycle time
  - Isolate architectural characteristics at quantum level: building non-functional requirements as fitness functions and isolate architectures, different architectures per quantum for example in micro service architecture
  - Adaption vs evolution: adaption is not focusing on new behavior, evolution requires fundamental changes
- Why Not?
  - Can't evolve a ball of mud: project's feasibility, make it modular first than try to evolve
  - Other architectural characteristics dominate: other aspects might be valued higher than the evolutionary aspect
  - Sacrificial architecture: simple versions to investigate markets or prove viability, these architectures are not meant to be evolved but the be replaced
  - Planning closing business soon: if business continuity is not guaranteed don't invest money in something that can evolve as it will never evolve over time.

Convincing others

- The business case: software often seen as a cost center, overhead
  - Moving fast without breaking things: incremental changes without consistent breakage
  - Less risks: modern practice
  - New capabilities: allows for hypotheses driven development