Architecture Vision

1 Business Drivers

Business Drivers inform Technology Vision which feed Technical Themes

Business Drivers	Technical Vision
New Features (lead)	Extensibility
Time to Market (speed)	Software Architecture Efficiency
Availability (work)	Availability/Reliability
Agility (react quickly)	Continuous Delivery

2 Technical Vision

2.1 An Efficient Software Architecture

Expose Single Systems of Record - Keep the system DRY

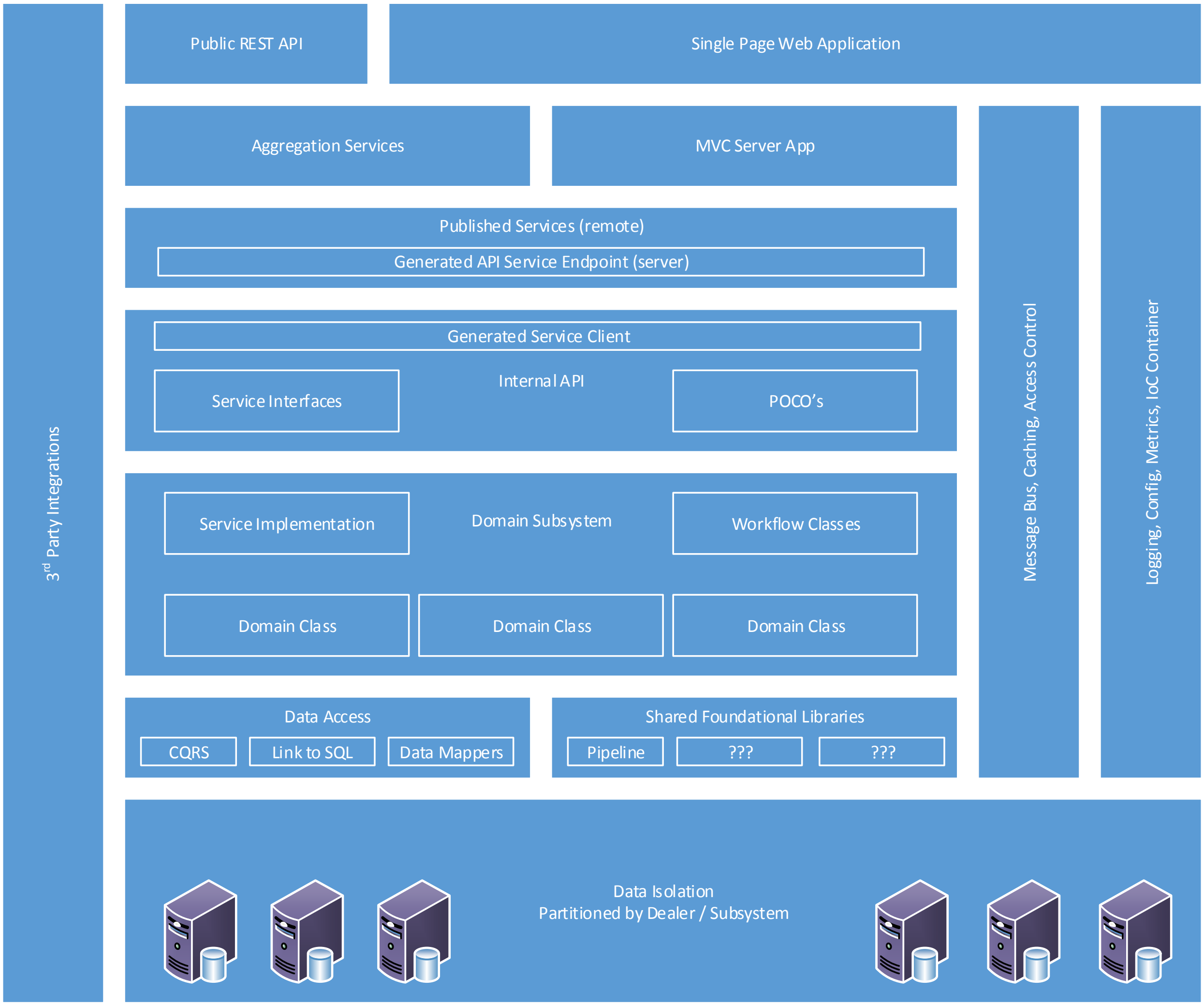
- Increase System Modularity
- Reinforce domain driven engineering practices
- Only one set of domain services per business function
- Partition data by related domain function

Create a suite of cooperative but discrete subsystems – Decouple systems and isolate areas of change

- Independently deployable subsystems
- Free teams to release software without fear of side-effects
- Only release the software that has changed
- Quickly Isolate production issues by application area
- Provide targeted scaling strategies on a per-subsystem basis

Infrastructure Focus - Provide teams with the best tools for the job

- Enterprise Service Bus
- H/A Caching Cluster
- Centralized Log Storage
- Built-In Metrics Collection
- Real Time System Health Reporting & Alerting



A Straw-Man layered architecture diagram.

2.2 Extensibility

Service Oriented Architecture – Expose core business functionality as service endpoints

- Introduces code mobility and allows for location transparency. This greatly increases deployment flexibility.
- Allows support for multiple clients improving interoperability and allowing business functionality to be made available at different tiers of an application. Decouples domain services from the technology used to develop it keeping the technology organization nimble and able to adopt new tech stacks as needed.
- Improves the ability to compose services in new ways and react to new business opportunities.
- Deploying services in a clusters and balancing load across the cluster improves availability and simplifies scale out as a method for increasing transactional capacity.

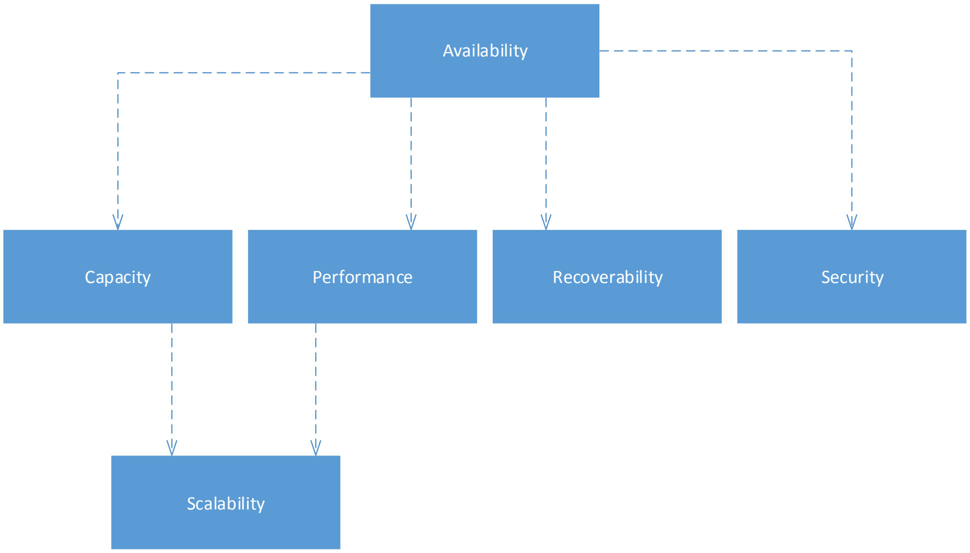
Domain-Driven Development – Focus development in areas of cohesive business functionality

- Helps keep related code together

- Subsystems to become the system of record for important business concepts
- Helps development teams organize around core business functions and gain valuable domain expertise
- Naturally creates code ownership

2.3 Availability / Reliability

Availability



Scalability

- Database Segmentation
- Database Virtualization ready
- Caching ready

Security

- Input Validation and Filtering
- Fine grained authorization
- SSL Everywhere
- Mature Key Management

Reliability

- Isolated subsystems
- Enhanced Instrumentation/Diagnostics
- multi-region deployment
- multi-availability zone deployment

2.4 Continuous Delivery

Release from master branch

- Code in the master branch is production ready

- Merging into the master branch indicates “intent to release”
- Code always rolls forward; never back

Dedicated build servers for initiating process steps

- Use Jenkins to react to code changes and initiate build/test/package/publish/deploy
- CI servers can be clustered to distribute continuous delivery workload
- Windows CI servers for windows stuff / Linux CI servers for Linux stuff
- Exposes inefficiencies and costs – Human involved and bottlenecks become clear
- Provides incentives for a healthy software delivery

Employ IT automation for server spin-up and provisioning

- Zero-Downtime Deployments required
- Indirected Cloud Provider – Automation does not expect/bind to a specific cloud provider and can support any
- Software is “released” constantly so ceremony *and the risk* around releasing is reduced.
- Reduces waste and makes releases boring
- ship code faster
- deployment process times improve (no handoffs means no waiting between deploy process steps)
- repeatability and constant practice means fewer failed deployments
- mean time to repair increases substantially
- enables ability to release specific features to specific customers
- requires the build-out of operational and software architectural infrastructure

Improve Test automation

- Manual testing should only occur as a means to produce automated tests
- Stories are not considered “done” until the automated unit/integration/acceptance are done
- Testing is not a QA-only responsibility

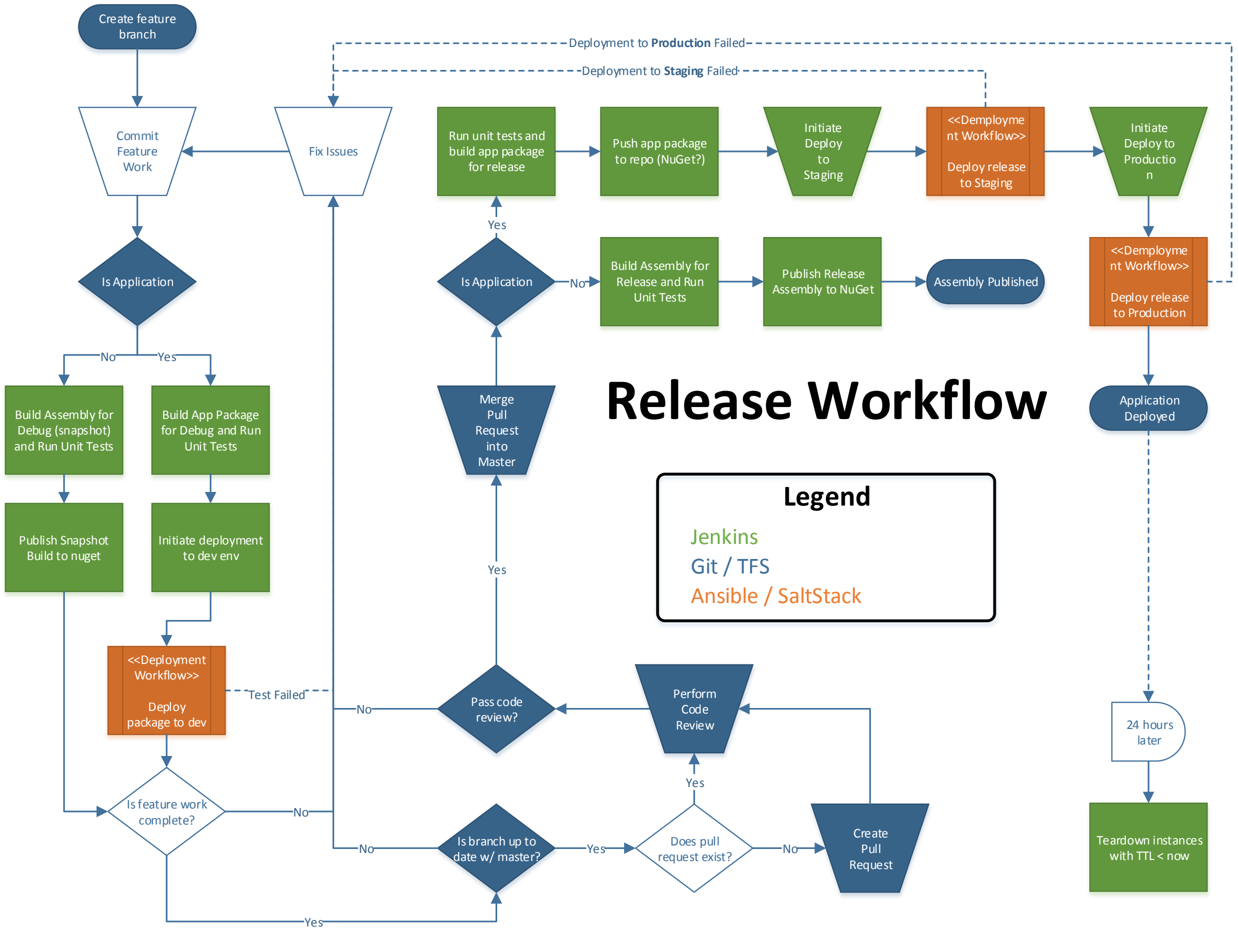
Eliminate Manual Intervention

- Full automation enables Faster reaction times
- Allows quick reactions to external and internal change
- Quickly resolve priority 1 security issue

Compartmentalized Deployment

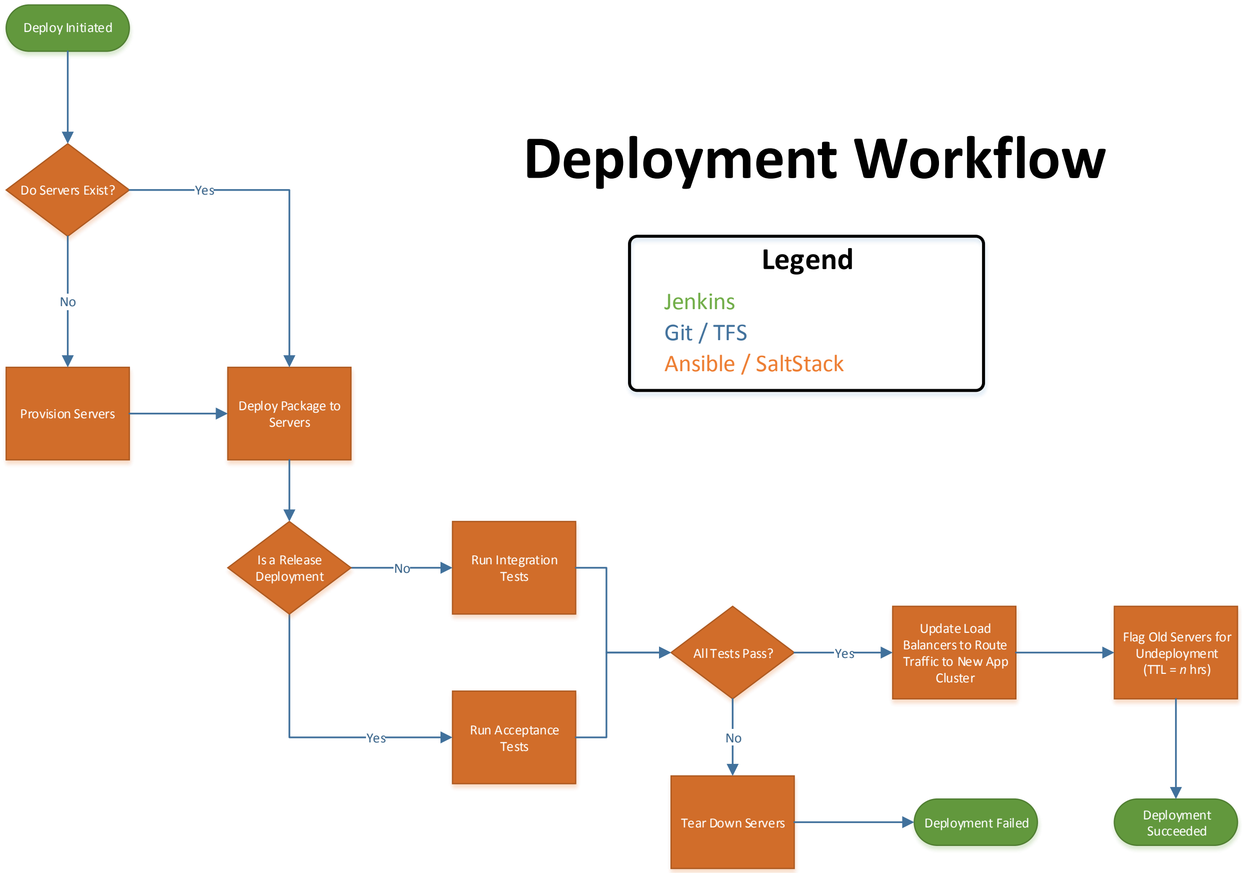
- Don't deploy everything everywhere, deploy with intention
- Deployment configuration lives with the source code and uses desired state configuration, IT automation, configuration management, cloud provision and multimode orchestration.

Ref: http://blogs.atlassian.com/2014/07/skeptics-guide-continuous-delivery-part-1-business-case/
Ref: http://puppetlabs.com/sites/default/files/2014-state-of-devops-report.pdf



Release Workflow

Create feature branch
Deployment to Production Failed
Deployment to Staging Failed
Commit Feature Work
Fix Issues
Run unit tests and build app package for release
Push app package to repo (NuGet?)
Initiate Deploy to Staging
<<Demployment Workflow>>
Deploy release to Staging
Initiate Deploy to Production
Yes
Is Application
No
Build Assembly for Release and Run Unit Tests
Publish Release Assembly to NuGet
Assembly Published
<<Demployment Workflow>>
Deploy release to Production
Is Application
No
Yes
Build Assembly for Debug (snapshot) and Run Unit Tests
Build App Package for Debug and Run Unit Tests
Merge Pull Request into Master
Application Deployed
Publish Snapshot Build to nuget
Initiate deployment to dev env
Yes
Legend
Jenkins
Git / TFS
Ansible / SaltStack
<<Deployment Workflow>>
Deploy package to dev
Test Failed
No
Pass code review?
Perform Code Review
24 hours later
Yes
Is feature work complete?
No
No
Is branch up to date w/ master?
Yes
Does pull request exist?
No
Create Pull Request
Teardown instances with TTL < now
Yes



Deployment Workflow

3 TECHNICAL THEMES

Dependency Injection

Invert the direction of dependency for keystone, proxy-able, & extensible software artifacts through the use of an Inversion of Control container

Factories

Support polymorphic binding of services and portable types via factory Methods & Abstract Factories

Domain Model

Separately manage state and behavior at the boundary of the subsystem. Delegates to data access to the CQRS framework for data storage and retrieval. These are Service Interfaces, Service implementations, Portable Types

Portable Types

Data objects that represent domain concepts. Lightweight, atomic and serializable to support movement across system boundaries and fine grained caching. Forms the contract in a service request or response.

Subsystem Isolation

Each subsystem is its own application independent of the others. An entire web cluster could be created to deliver a single high load subsystem. Examples of such systems are: Ordering, fsw.com, Product, Search

Localization
Enabled localization and formatting of strings, dates, and numbers. Separate application resources that require translation from the rest of the code. Resource strings are isolated to their own resource assemblies. This is a small incremental cost with potential for high value by building our software to easily support multiple languages and time zone.

Data Sharding
Introduce ability to treat many physical databases as a single logical data source. This is about laying the groundwork for massive future growth. This is another area where a small amount of upfront work high future yield by creating the penitential for > 100x database scale. Planning to shard by 'atomic unit' helps us understand out data better which further improves the software we build.

Enterprise Messaging
Provide notification of all state change events; leverage messaging infrastructure for guaranteed delivery of events to subscribers. RabitMQ is excellent, proven software that works but benefits from a formalized strategy for interaction and expected/acceptable uses. Consider looking into SubPub to provide http-based service oriented abstraction to the underlying RabitMQ messaging system.

Consistent APIs
Provide uniformity of interface format, annotation, and style, ensure ease of maintainability. Well-structured modular design promotes API predictability. Consider introducing coding standards and style enforcement as well as introducing code reviews as part of the standard release process.

Cohesive Services
Service responsibilities are highly related and focused. These responsibilities should targeted to one functional area. Service methods should have a clear, single purpose. We will know we have accomplished this because clients will not be forced to depend on methods they don't use → No "fat" interfaces.

Indirected Coupling
Indirect dependence promotes modularity and minimizes the impacts resulting from changes to 3rd party components. Introduce Wrappers and frameworks around 3rd party components to ensure a level of "control" and allow for competent replacement without pervasive impact on the entire system.

Fine Grained Build Artifacts
Assemblies should be isolated from the impacts of change, because responsibilities (axis of change) are not distributed across deployed units. Each project produces an assembly which should be published to NuGet (or some assembly repo). Windows applications should be published to Chocolatey. Node.js modules should be published to NPM. Python modules should be published by PiPy. Etc…

Managed Stability & Instability
Depend in the direction of stability. Changeable/Flexible classes / assemblies are not tightly bound to each other. Prescriptive dependence on interfaces when instable. Dependence on concrete classes when dependency is stable. Remote services can be an excellent substitute for interface coupling.

Instability can be desirable, otherwise things would be unchangeable, but we mitigate that instability by depending on Interfaces. Log4Net might be an example of a stable object which would be acceptable to depend on directly.

Running with Least Privilege

Reduces potential for damage from a compromise. The account is not as capable of inflicting damage; privilege escalation is mitigated. DB Connection strings should use an account with least privilege and which as access to only the schema/database necessary to accomplish the behaviors of a specific domain. IIS and ASP.Net worker process should also run under local user accounts having very little access to anything, including other network resources.

Defense in Depth

Prevent attacks by implementing multiple counter-measures. Perform token tampering detection and validation; segment logical environments into physical environments; consider introducing physical network partitions by application layer (ie: ui/middleware/data). Encrypt data at rest and in transit.

Input Filtering

User input should be considered hostile necessitating the filtering of malicious elements and escaping suspicious elements. Ensure the system performs html escaping on all user and external system input.

Instrumentation

Ensure capacity planning is predictable, system health is monitored in real-time, and troubleshooting is rapid.

Private vs Internal vs External API's

Private APIs are intended to be used only by the owners of the APIs themselves. Internal APIs are intended to be used between sub-systems and applications (crossing team ownership boundries). External APIs are introduced as insulation against change and to provide friendly public façade's targeting specific customer needs.