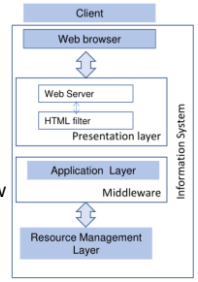
Latency
IAAS,PAAS,SAAS
**Cloud-native Fts**
- Immutable infra
- Microsvcs
- API driven
- Service mesh
- Containers
- Dynamically mgd
**Development**
CI/CD
DevOps
Serverless
**Stack**
- Infra
- Provisioning
- Runtime
- Orchestration, mgmt
- App defn, devt
- Observability
**Deployment**
- Integration
- Content delivery
- Heterogeneous plats
- Coordination
- Security
**Containers**
- Runtime with code (interop, portable)
- Dep, change mgmt
- Env mgmt
- Reproducible
- Isolation, security
- Orchestratable
**Orchestrator (K8s)**
- Provisioning,
- deployment (coordn, fault tol)
- networking
- scaling
- availability
- lifecycle mgmt
**Serverless (AWS λ)**
- via containers
- Func-AAS (FAAS)
- Event-driven execution model
- Stateless, ephemeral, managed by PAAS
**SDLC**
- Waterfall
- Agile

**CI/CDel/Dep**
Del: manual dep
Dep: auto dep
- Low risk release
- Faster to mkt
- Higher qual
- Reliable
- Scalable
- Lower cost
**Devops**
- Commit, build, test, release artifacts,
- CI/CD/ monitoring/ communication
**Requirements**
Elicit, Analyse, Specify, Validate
**SWE Req Spec**
1. FRs
2. NFRs
3. System Req
4. Constraints
5. External Interfaces
**SRS Qualities**
1. Accurate
2. Complete
3. Modifiable
4. Ranked
5. Testable
6. Traceable
7. Unambiguous
8. Verifiable
9. Valid
**Business Req**
E.g. Reduce costs by 25%
**User Req**
- As a (user), I can (action) so that I can (action)
- [ref] The system supports png and jpg
**FR:** what system does
**NFR:** how well works (reliability, efficiency, usability)
**System Reqs**
Hardware
Systems

**External Interfaces**
Connections to outside world
**Constraints**
e.g. Backward compatible, mem usage
**External Quality Attr**
UX, observable
1. Availability
2. Safety
3. Deployability
4. Usability
5. Performance
6. Security
7. Compatibility
8. Reliability
9. Installability
10. Integrity
11. Robustness
12. Interoperability
**Internal Quality Attr**
Felt by devs, maintainers
1. Testability
2. Efficiency
3. Modifiability
4. Scalability
5. Portability
6. Reusability
7. Verifiability
8. Maintainability
**Security**
- privacy
- authentication
integrity
**Performance**
- Response time
- Throughput
- Data capacity
- Concurrent users
- Predictability
- Latency
- Behaviour when overloaded
**Scalability**
- hard/software
- Vert scaling (bigger)
- Hori scaling (more)
**Availability**
- up time/total
- Five 9s=99.999%
- needs redundancy

**Usability**
- UX
- num clicks to do X
- ease of learning/use
**Traceability**
- type, ID
- descr
- priority
- etc
**Validation**
- biz-relevance
**Verification**
- requirements are correct (complete, correct, feasible, prioritised)
**Control Flow**
**Data Flow**
**Call and Return**
- movement of control
**Message**
- Sync
**Event**
- emitted for listeners
- immutable
- ordered
**Architecture Decomposition**
Hori slicing by layers
Vert slicing by fts
**Cohesion Types**
Functional, Layer, Comms, Seq, Procedural, Temporal, Utility
**Coupling Types**
content, common, control, data, external, temporal, import
**Technical Partitioning**
- SoC
- Aligns with expertise
**Domain Partitioning**
- align with domain
- logical components map to problem

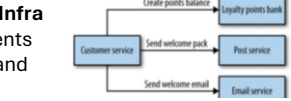
**Layered Architecture**
- monolith + technical partitioning
- n-tier architecture with different layers, (+) perf optim, min context switching (-) less modifiable, new function changes multiple parts
**Modular Monolith**
- monolith + domain partitioning
**Event-Driven**
- Distributed + Technical partitioning
**Microservices**
- Distributed + Domain Partitioning
**Pipe and Filter**
- Data enters and flows through components as a stream (incremental computation).
- Independent filters
- Data source → filter → data sink (via pipes)
- Useful for batch processing
**MVC**
- V: UI (http response)
- C: coords M and V (handles http, select model, preps view)
- M: business logic + persistence (server)
V → C: user actions
C → V: select new view
C → M: update model
M → V: M triggers V update
V → M: query for state
**Benefits of MVC**
- SoC: modularity btwn app state and processing
- Extensibility: new V/C pair can be added
- Reduced comm complexity
- Testable/mockable
**Single Page App (SPA)**
JS program in browser w/o refreshing
- less perceived latency



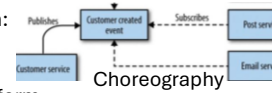

**Architecture Diags**
- big picture map
Software system
Container is the context or boundary which executes code or stores data
Components group related functionality behind interface
**REST**
- REpresentational State Transfer
- stateless
- self contained
- cacheable for network efficiency
- application is layered
- Uniform API interface
-- Via resource identifiers
**Pros and Cons**
(+) less tightly coupled
(+) scalable, usable, accessible, mixable
(-) repeated queries
(-) stale cache
(-) unspecific data (cf GraphQL)
**DDD**
Interactions between contexts model the interactions in the subdomains
**Domain:** the problem space
**Sub-domain:** component of main domain, focusing on specific set of resp; reflect biz org struct

**Bounded Context:**
Highly cohesive boundary, includes input, output, events, reqs, procs, data models relevant to sub domain.
**Aggregate**
- Cluster of related objs treated as a single obj for data changes
- High cohesion, derived from business reqs
- Xact boundary: all changes succeed or none
- Consistency boundary: externals read state, or execute public interface methods
**Aggregate Root**
- parent entity of cluster
**DDD Shared Kernel Collaboration**
Two contexts devt independently but end up overlapping
**DDD Upstream-Downstream Collab**
(provider-consumer r/s)
**DDD Comms**
Integration between BC are sync or async
**Event Storming**
- Domain events are simply some sort of event that occurs in the domain & that are relevant.
- A command causes an event
- Find aggregates
- Policy (WHEN event THEN command)
Cmd-Ag-Ev → policy → Cmd-Ag-Ev
E.g. when the payment card is submitted, then confirm order.
- Draw boundaries
**Database-server-per-Service**
+ loose coupling, scaling at db level
+ easier to replace underlying DB
- expensive
Data Independence: private schema/tables enforce data security at db levels e.g. db users
**Data Delegate Pattern**
service which all others look to for a db access
**Data Lake pattern**
read-only, queryable data sink. Shared space containing (copy of) data from concerned microsvcs
**Saga (Data) Pattern**
- Every action has a compensatory action
- If error on stage 4, compensate earlier stages
- Note not ACID
- Sequence matters
- Hard to compensate actions (eg notifs) to end
**Event Sourcing (Data)**
- Events: Source of Truth
- System state: projected from series (use snapshots for perf)
- append mutations to Event Store (ID, eventType, data)
- notify multiple subscribers to build projections
- for scalability, time travel, auditability, traceability
**Command Query Resp Segregation (CQRS)**
Cmd mutations return nth
Queries read, return value
- Separate read/write models (imbal R/W)
- E.g. write to Kafka event stream via KSQL, materialise as precomputed query to update (multiple) DB
- Eg read model is adiff table or denorm Mongo w push/pull from EventStore
**Service Instance per Host/VM/Container**
**Devops Practices**
- Local workstations to hosted infra
- Provision env, deploy to hosted
- Cheap for new env

**Immutable Infra**
All components reprovision and recreate
**Infra as Code**
Orchestration: K8s
Config: Ansible/Terraform



Orchestration



Choreography

**Svc Collab: Orchestration vs Choreography**
- central brain vs inform each its job
**Svc Comm:**
Sync/Async IPC
Req/Response
- sync/async
Notif: (one way)
Event-driven
Real time data streams, loose coupling
**API Gateway**
- single entry pt into system
- tailored API
- other responsibs like authenticates, monitoring, LB, caching
**Backend for Frontend (BFF)**
- diff gateway for diff platform (web, mobile)
**Svc Discovery**
API Gateway needs to know IP addr, port of each microSvc.
Client-Side Disc
- Svcs register w svc registry
- Queries registry
- LB algo to select
Svr-Side Disc
- Svcs register w svc registry
- Client requests svc via router/LB.
- LB queries svc reg, routes requests to service

**Svc Registry Pattern**
- Svc Registry knows instances, addr, port.
- Instances register on startup, dereg on shutdown
**Initiating Event:**
from end user which kicks off business process
**Derived Event:**
internal s/w event triggered by init event
**Event Structure**
- Unkeyed: stmt of fact
- Entity: K-V pair
- Keyed: not an entity, used for partitioning data stream for data locality within a single partition
**Async Comm**
- more responsive
- more available
- more complex error handling
**Event-Driven Arch**
- Producer (pub), Broker, Consumer (sub)
- Bounded Contexts impt for microsvcs, but nice-to-have
- Built on event processing i.e. responding to sth that happened, and trigger more events (Microservices based on responding to requests)
**Event Broker**
- recvs events, holds in queue/partition, sends to consumer
- immutable, ordered append-only

**Event Broker Data Handling**
To consume events from a partition
Seek to ID of last offset
Scans sequentially, reading msgs in order and save its recording new partition offset,
**Event Broker Characteristics**
- Immutable, Ordered, Indexed, Partitioned, Persistent, Replayable
**Scalable**
- Reading and writing can be done in parallel with events in partitions over all the machines provided by the broker.
- Scaling: add new machines and rebalance.
**EDA Pros**
(+) Perf, maintainable, scalable, evolvable, fault tolerant
(-) Complex, Hard to test
**Scalability**
- Increase capacity
- Optimise available resources
Scale up: (vert scaling) bigger CPU
Scale out: (hori scaling) replicate srvr
**Load Balancer**
- Chooses svc replica to process req
- Relays response
**Session Store**
- Data representing user session might need to exist somewhere
**DB Caching**
- stale/invalidation/TTL
- e.g. Redis
**DB Read Replicas**
- distributed database
- primary DB with read replicas (changes asynchronously replicated)
**DB Hori Partitioning (sharding)**
Hash(PK) = partition

**DB Vert Partitioning**
- columns are split
**Overall**
- more storage nodes
- partition data across disks
**Swim Lanes**
- boundaries placed, split by regions
- faster response time regionally
- fault contained to swim lane
**Scale Cube**
X-axis: duplication replicas behind LB
Y-axis: func decomp split app up
Z-axis: data partitioning
**Synch Request-Reply Pattern**
HTTP
**Async Request-Reply**
HTTP
1. query API endpoint
2. poll status endpoint
3. query resource URI
**Topic vs Queue**
topic: one-to-many
queue: single receiver, competing
event: pub owns event + channel (topic)
msg: recv owns msg + channel (queue)
**Kafka**
add as many brokers as required
**Adv MQ Protocol (AMQP)**
- p2p msg protocol
- msg published to exchanges, exchange distributed to queues
- broker deliver msg to subs / consumer pull from queue
**RabbitMQ (AMQP)**
Binding: link from Q to exchange
Routing key: how msg routed to queue
- producer publish to exchange, exchange routes to Q, msg stays in Q until handled

**Exchange Types**
Direct: exactly to queue
Fanout: to all queues
Topic: match regex
**Persistent Comm**
(store-and-forward)
- msg stored at intermediate until next node is ready
**Transient**
- msg buffered for fixed period before discard
**Enterprise Service Bus**
- central bus to connect apps and data
**Messaging Broker (Message-Oriented Middleware MOM)**
JMS: message standard
AWS SQS: messages are replicated
**Message Construction**
- header containing type of info, origin, dest, size and other metadata
- payload
**Message Channel (MQ)**
Return addr: tell replier where to send the reply
Correlation ID in reply specifies which request the reply is for
**Request-Reply chaining**
- A req causes a reply, reply is another req, etc
- Chaining is useful if app wants to retrace the path of msg from the latest reply back to the original request.
**Invalid Msg Channel**
- handle erroneous msgs
**Dead Letter Channel**
- handle failed msgs e.g. due to network, downstream failure, logic errors, causing unconsumed msgs in Q
**Datatype Channel** (for each data type)
i.e. all msgs on this channel are of same type (e.g price quote)
e.g. RMQ use routing key to determine which Q

**Pub-Sub**
all msgs in topic recved by all subscribers
**Message Router**
- consume msg from a ch and reinserts into a diff ch
Simple Router: from one inbound ch to one or more outbound ch
Composed Router: combine multiple simple routers to create more complex msg flows
Context-Based Router: decides msg dest based on context (usu performs LB, test or failover): send msgs to other ch/dest via detour
Content-Based Router: examines msg content and routes based on data—requires knowledge of all receipients
Message Filter: a subtype of content-based router that discards undesired msg
**Content Based vs Pub-Sub+Filter**
cb: 1 consumer
ps: >1 consumer
cb: central ctrl/maint, predictive routing
ps: distributed ctrl/maint reactive routing
cb: router needs to know participants, update when changed
ps: no knowledge needed
cb: biz xacts eg orders
ps: notifs, info msgs
cb: more efficient w Q-based ch
ps: more efficient w pub-sub
**Message Splitter**
split msg into smaller parts e.g. Order into multiple Order Item (to be routed via Content-Based Router)

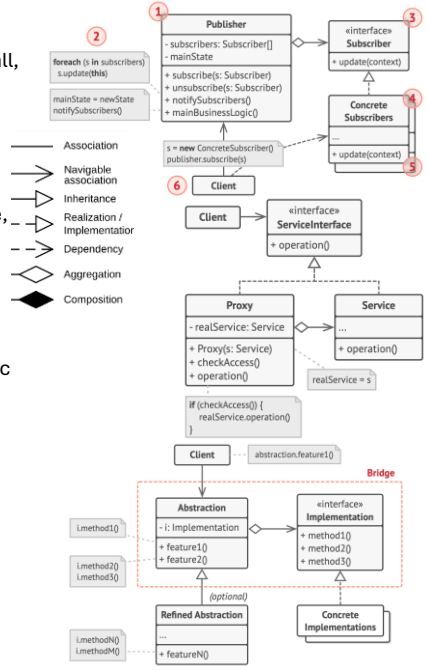
**Message Aggregator**
- Groups multiple msg tgt, then
- publishes a single aggregated msg to output ch
**Message Scatter-Gather**
- broadcast to number of participants, then aggregates replies
- via splitter + aggregator
**Message Translator**
- convert btwn fmts
A Canonical Data Model may be used that all data fmts convert to
**Message Endpoints**
- the interface btwn app and msg system (custom to each)
- either to send or recv, but not both
- channel specific
Consumer Endpoints
Polling Consumer
Event-Driven Consumer
**Message Patterns Summary**
1. Message Construction
2. Message Channels
3. Message Routing
4. Message Transformation
5. Message Endpoints
**OO Behaviours**
**Bridge Pattern**: abstract two diff interface hierarchies into composition
**Proxy Pattern:** proxy between base obj and client, proxy handles NFR (e.g. lazy init, logging, access, caching)
**GoF Design Patterns**
Creational Patterns: e.g. Factory
Structural Patterns: structure to obj r/s e.g. Façade
Behavioural Patterns: how obj interact with each other
**Adapter Pattern**
translate fmts/Canonical

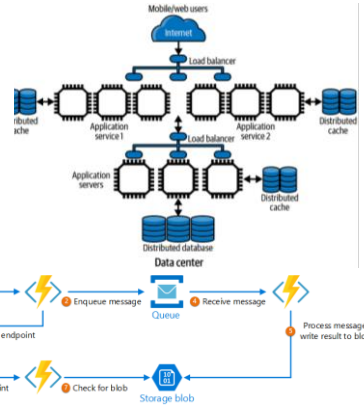
**Façade Pattern**
- unified interface to a set of interfaces
- E.g. API gateway
**Observer Pattern**
Observers register to subject. Events in subject notify observers (push data snaphsot).
Observers may call subject to pull info.
Categorised pull Model: Observers register only with specific changes
Categorised Push Model: Push state only to interest observers
Mediator Pattern
Talk to middleman.
EventBus is an architectural mediator
**Data Transfer Object (DTO) Pattern**
a struct to group data to reduce remote calls, reducing RTT and network overhead
**SOLID:**
- Single Resp Principle
- Open-closed Principle (open for extension, closed for modification)
- LSP (subclasses substitutable)
- Interface Segregation Principle (don't implement interface unnecessarily)
- Dependency Inversion Principle (use abstractions like interface)
**Non-solid**
SoC, High Cohesion (relevant together), Loose Coupling (don't depend), Info-hiding, Encapsulation

**External quality attr:**
- Availability: Uptime
- Installability: new install, updates, recover
- Integrity: Data loss
- Interoperability: interaction w other systems
- Perf: latency, response, throughput, concurrent users
- Reliability: failures
- Robustness: error handling/fault tol
- Safety: parental ctrl, etc
- Security: malware
- Usability: UX
**Internal Attr**
- Efficiency: PU usage
- Modifiability: understood, changed, extended
- Portable: how much code change for ios-android
- Reusable: modularity
- Scalability
- Verifiability/testability

Shift Left Testing
Observer allows open/close, adding/removing at runtime, but notification in random order

Façade, mediator cause god objects







DDD
1. Entities in Domain
2. Clustering to aggregates
3. Business Domain to BC
4. Microservices

Event storming
1. All events
2. Events in sequence
3. Model ecosystem
4. Aggregates
5. Bounded context