Application Architecture | AppArch

Zusammenfassung

CONTENTS

1. REQUIREMENTS & CONTEXT

Software Architecture can be defined in different ways with different emphasis. The most concise definition is by ***ISO 42010:***

> *"The fundamental organization of a system is embodied in its* **components***, their* **relationships** *to each other, and to the environment, and the* **principles** *guiding its design and evolution."*

Or in short: How different pieces work together and what principles for further developments are in place. Software architecture is used to improve maintainability, scalability and collaborative work in a software project.

Difference between IT-, Software- and Application Architecture:
- ***IT*** is the ***most general*** *(hardware and software)*
- ***Software*** is the ***high-level structure*** *(independent of actual technologies, microservices vs. monoliths, NFRs, interoperability etc.)*
- ***Application*** is the ***logical design*** of a software *(actual technologies, design patterns, UX, dependencies)*.

Application architecture is a ***subset*** of Software architecture. A ***Software Architect*** needs to understand requirements, acts as a mediator between stakeholders and must make technology decisions with foresight.

Design Challenges on Enterprise Software Projects
- ***User and channel diversity:*** Different actors, data/request volumes, technologies
- ***Process and resource integrity:*** Data and processes must be trustable
- ***Integration needs due to heterogeny:*** Software must integrate in different systems with different data formats
- ***Complex data/domain models and processing rules:*** Implement business rules in a maintainable way

The 10 Keys to Success
Define what successful Architects should consider in their work.

1. ***Understand end-to-end development:*** Follow a repeatable process to get the output of a task
2. ***Understand your role:*** Architects must review existing architecture, share knowledge with peers and communicate the value of their solutions
3. ***Manage risk & change:*** Derive architectures iteratively via abstract-and-refine
4. ***Communicate with stakeholders:*** Document architectures to capture design analysis and results, preserve the reasons behind decisions
5. ***Reuse assets:*** Use different types of assets that already exist and apply them
6. ***Right-size your involvement:*** Select relevant viewpoints, hide unnecessary details
7. ***Influence the requirements:*** Ensure tradeoffs are negotiated, predict system behavior and time/cost of development
8. ***Derive solutions from business needs:*** Design architectures based on the business you're building the application for
9. ***Refine solutions based on technology:*** Specify technical designs and guide development
10. ***Appreciate the broader context:*** Align your work with the bigger picture, document the system context

1.1. REQUIREMENT MODELLING STRATEGIES

Good requirements for software projects ***address the needs of the stakeholders***. The following models try to give you guidance on how to formulate such requirements.

1.1.1. Architecturally significant requirement (ASRs)

Requirements that are architecturally significant have a ***measurable effect*** on an architecture. They help to prioritize technical issues or requirements quickly, so that architecturally significant issues are addressed at their most responsible moments.
To ***determine*** if an issue might be ***significant***, we can check if it addresses one of the following topics:

- Number of users
- Availability
- Lifetime
- Extendability
- Regulations
- Laws
- Safety
- Security

A ***more detailed approach*** is the ***ASR Test*** *(The points six and seven are very subjective)*.

1. **RC-1 – High Business Value and/or Risk:** The requirement is ***directly associated*** with it.
2. **RC-2 – Stakeholder Concern:** The requirement is a concern of a particularly ***important stakeholder*** *(for instance, the project sponsor or an external compliance auditor)*.
3. **RC-3 – QoS:** The requirement has runtime ***Quality-of-Service (QoS) characteristics*** *(e.g., performance needs)* that ***deviate*** from those already satisfied by the evolving architecture ***substantially***.
4. **RC-4 – External Dependencies:** The requirement causes ***new*** or deals with one or more existing external ***dependencies*** that have ***unpredictable***, ***unreliable*** and/or ***uncontrollable*** behaviour.
5. **RC-5 – Cross-Cutting:** The requirement has a cross-cutting nature and therefore ***affects multiple parts of the system*** and their interactions; it may even have system-wide impact.
6. **RC-6 – First-of-a-kind:** The requirement has a first-of-a-kind character: e.g., the team has ***never built a component*** that ***satisfies*** this particular requirement.
7. **RC-7 – Past Problems:** The requirement has been ***troublesome*** and ***caused critical situations***, ***budget overruns*** or ***client dissatisfaction*** in a previous project in a similar context.

Example of a ASR Test

The following paragraph gives an ***example analysis*** of ***requirements*** regarding their ***architectural significance***. A single requirement is mapped to the different points. Make sure to always ***justify the answers*** in order to make them understandable. The score is either Low *(L)*, Medium *(M)* or High *(H)*.

Requirement	*Score*	*Mapping*	*Explanation*
Dynamic and fair pricing engine: *The platform must provide a controllable but dynamic pricing system. It should suggest fair prices based on existing games and data (e.g. usage). However, it should be possible to respect the wishes of game developers and allow for dynamic and automated adjustments based on usage and demand.*	H	– RC-1 *(important part of business model and revenue)* – RC-2 *(important for multiple stakeholders, namely the developers as well as the gamers)* – (RC-4) *(might need external libraries, AI tools?, or data sources)*	Pricing influences business model and revenue; affects many components *(checkout, listing, recommendation)*. External data dependencies and correctness requirements make it architecturally significant.
Age & Payment Compliance: *The platform must ensure compliance with age restrictions and safe in-game purchase flows (no dark patterns, parental controls).*	H	– RC-1 *(legal/business stakeholders)* – RC-2 *(user-visible)* – RC-4 *(regulatory dependencies, 3rd-party payment providers)*	Legal/regulatory impact *(age verification, payment compliance)* creates business risk and may block the service in jurisdictions — high priority.
Discovery/Search & Recommendations: *Gamers must be able to discover games via search, filters and personalized recommendations within acceptable latency.*	M-H	– RC-2 *(user experience)* – RC-3 *(performance / scale)*	Core to product value *(finding games)* quality affects retention. If personalization is central, architectural patterns *(indexing, ML infra)* are required.
Community Features: *The platform should allow gamers to write short public reviews and give star ratings for games.*	L-M	– Maybe RC-1 *(stakeholder concern)*, but secondary – Partially RC-2 *(user-visible behavior)*, but doesn't affect critical flows	Reviews and ratings can enhance user engagement and discovery, but they are not mission-critical to the business model or compliance. They can be added incrementally, with well-known architectural solutions *(e.g., simple CRUD + moderation pipeline)*.

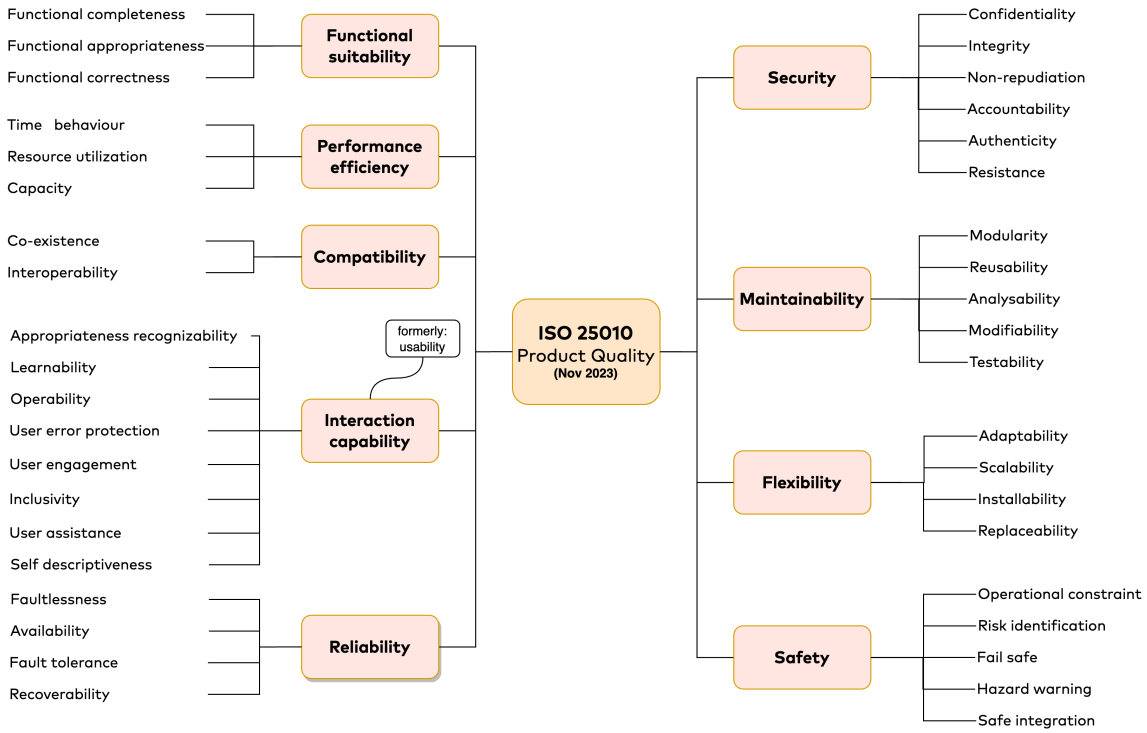
1.1.2. Architectural Significance of Decisions

The following questions can be used to classify ***the architectural significance of decisions*** that justify whether and how a requirement or other concern is met:

1. Is the decision ***hard to change*** later?
2. Is the decision ***expensive*** to ***implement/execute*** upon?
3. Are ***demanding qualitative requirements*** stated? *(e.g. high security level, high availability, high performance)*
4. Are requirements ***difficult to map*** to existing solutions, experiences?
5. Is the ***experience*** *(of the team)* in the solution space ***weak***?

1.1.3. ISO-25010 Software Product Quality

Most management models base their quality attributes on ISO 25010. It has ***nine characteristics*** with several attributes. NFRs are usually built around these categories. Shown below is the 2023 version of the standard.



1.1.4. FURPS

FURPS provides a way to define requirements by non-functional stories, and also provides a good way to categorize such needs. The requirements generated by FURPS are called ***Architectural Significant Requirements (ASR)*** and are often assigned an ID.

- **Functionality:** Represents the ***main product features*** that are familiar within the business domain of the solution being developed. The functional requirements can also be very ***technically oriented***. *(FR)*
- **Usability:** Includes looking at, capturing, and stating requirements based around ***user interface issues***. Things such as ***accessibility***, ***interface aesthetics***, and ***consistency*** within the user interface *(NFR)*
- **Reliability:** Includes aspects such as ***availability***, ***accuracy***, and ***recoverability***. *(NFR)*
- **Performance:** Involves things such as ***throughput of information*** through the system, ***system response time***, ***recovery time***, and ***start-up time***. *(NFR)*
- **Supportability:** Here a number of other requirements can be specified, such as ***testability***, ***adaptability***, ***maintainability***, ***compatibility***, ***configurability***, ***installability***, ***scalability***, ***localizability***, and so on. *(NFR)*

FURPS+ also includes design constraints, implementation constraints, physical constraints and interface constraints.

But FURPS does have its problems, see Chapter “Architectural Viewpoints” (Page 24).

Example of FURPS+

Criteria	*Description and Rationale*
Functionality (ASR-F1)	As a market operator, I want the platform to provide a pricing system that suggests fair prices based on game usage data, comparable market titles, and demand signals, while allowing developers to set their own preferred price ranges and apply manual corrections. The system shall combine automated adjustments *(e.g., demand-driven)* with developer-defined constraints, so that developers can override or restrict automated changes, and buyers always see a consistent and transparent final price at checkout. **Rationale / Significance:** Pricing is central to the business model and touches listing, checkout, analytics, and legal auditing.
Usability (ASR-U1)	A new developer should be able to create a developer account, publish a game with pricing rules and see the first price suggestion in the dashboard within 15 minutes after completing the guided onboarding *(including verification steps)*. The pricing UI must show a clear explanation of any dynamic adjustments *(source, timestamp, confidence)* inline next to suggested prices. **Rationale / Significance:** Low onboarding friction and transparent pricing are differentiators. UI + backend audit APIs must present explanatory metadata; this affects API design, audit logs, and front- end UX.
Reliability (ASR-R1)	All purchase flows that require age verification or payment processing must succeed end-to-end 99.95% of the time measured monthly *(excluding scheduled maintenance windows)*. Failures of those flows must not cause inconsistent financial states *(no partial charge without final confirmation)* and must be recoverable within 1 hour *(automated compensation or manual operator workflow)*. **Rationale / Significance:** Compliance and money flows are high-risk; correctness and recoverability strongly impact architecture *(transaction management, distributed sagas, compensations, monitor- ing)*.
Performance (ASR-P1)	Under normal operation *(up to 100k concurrent active sessions)*, the search & recommendations service shall return results within 350 ms *(95th percentile)* for typical queries; during promotional peaks *(up to 1M concurrent active sessions)*, tail latency *(99th percentile)* should remain below 1.5 seconds for cached queries and below 3 seconds for cold queries. **Rationale / Significance:** Discovery affects retention; sets constraints on caching, indexing, and autoscaling.
Supportability (ASR-S1)	Each major backend service *(Pricing, Search, Payment, User Management, Telemetry)* must expose standard health endpoints, structured logs (JSON), and traced spans *(distributed tracing)*. Any such service must be independently deployable and replaceable without requiring coordinated downtime of other services *(backward-compatible APIs for at least two major API versions)*. **Rationale / Significance:** Supportability requirements enable safe evolution, debugging, and operational resilience; they constrain API versioning and CI/CD strategies.

1.1.5. SMART NFRs

The ***SMART criteria*** are frequently used in ***project and people management*** but can also be applied to ***Non-Functional Requirement*** engineering. In this case, only ***S*** and ***M*** are used. The ***A***, ***R*** and ***T*** are more likely the concern of project management.

1. **Specific:** Which ***feature*** or part of the system *(component or user story)* should ***satisfy*** the requirement? Not all features need the same availability, not all quality properties are as cross-cutting as security.
2. **Measurable:** How can testers and other stakeholders ***find out whether the requirement is met or not***? Is the requirement ***quantified***? May include ***monitoring*** the requirement throughout the project.
3. **Agreed upon:** The goal must be ***realistic***, ***achievable*** and have a ***consent*** in the team.
4. **Relevant/Realistic:** One must be aware of ***why*** this should be achieved.
5. **Time-Bound:** There should be a ***clearly defined timeline***, including a starting date and a target date.

Example of a SMART Test

The ASRs from "Example of FURPS+" (Page 5) are tested to check whether they are ***Specific*** and ***Measurable***.

ASR-ID	*S?*	*M?*	*Rationale*	*Suggested Improvements*
ASR-F1	✓	✓	Defines scope *(pricing system, developer control)* and includes a measurable target *(5s)*.	Clarify override handling. *(e.g., "Developer changes applied within 1 minute, with audit log.")*
ASR-U1	✓	×	Clear actor/task *(developer publishes and sells)*, but "soon after joining" is vague.	Add time target *(e.g., "First sale visible in dashboard within 15 minutes of onboarding completion.")*
ASR-R1	✓	✓	Very concrete: 99.95% success and 1h recovery	Define monitoring method *(which failures count, how recovery time is measured)*
ASR-P1	×	×	Vague: "return results quickly" gives no specific feature scope or measurable target.	Add scope *(search/recommendations)* and latency thresholds *(e.g., "<350 ms for 95% of queries under 100k concurrent users")*.
ASR-S1	✓	~	Specific about observability *(health endpoints/ logs)* but "replaceable without downtime" is not well measurable.	Define measurable deployment window *(e.g., "Service replacement must complete in <15 min, with <5 min degraded functionality.")*

1.1.6. Agile Landing Zones

Establish ***three measurable values*** *(M in SMART)* rather than a single measure that might not be realistic *(R)* and impossible to agree upon *(A)*. Similar to ***release criteria*** but allow for ***tolerances*** in acceptable values. Allows for some ***flexibility*** in meeting goals without forcing you to accept unreasonable compromises. Can be time bound *(T)* individually.

"Minimal" defines the worst acceptable time frame, ***"target"*** the time frame to aim for and ***"Outstanding"*** the best possible case.

Response Time per Business Activity S	*Minimal Goal (less than)*	*Target (within)*	*Outstanding (within)*
Order fully processed	2 weeks	24 hours	3 hours
Relocation processed	3 weeks	2 weeks	1 week
Technician appointment scheduled	2 days	1.5 days	1 day
Address validated	10 seconds	3 seconds	1 second
Billing system configured	1 week	3 days	1 day

1.1.7. Quality Attribute Scenarios (QAS)

A quality attribute scenario specifies a measurable quality goal for a particular context.

NFR Specification Template

Scenario for Requirement xy

Scenario Synopsis		Summarize the NFR being specified
Business Goals		The business objectives supported by this quality requirement
Relevant Quality Attributes		Key quality characteristics this scenario targets
Scenario Components	*Stimulus*	The condition that affects the system
	Stimulus Source	Person/system that triggered the stimulus
	Environment	The condition the environment is in when the condition occurs
	Artifact	The thing affected and observed in this scenario
	Response	The desired system reaction to the stimulus
	Response Measure	Quantified acceptance criteria that makes the response testable/validatable
Questions		Open Points/assumptions to clarify with stakeholders
Issues		Known risks, constrains, conflicts

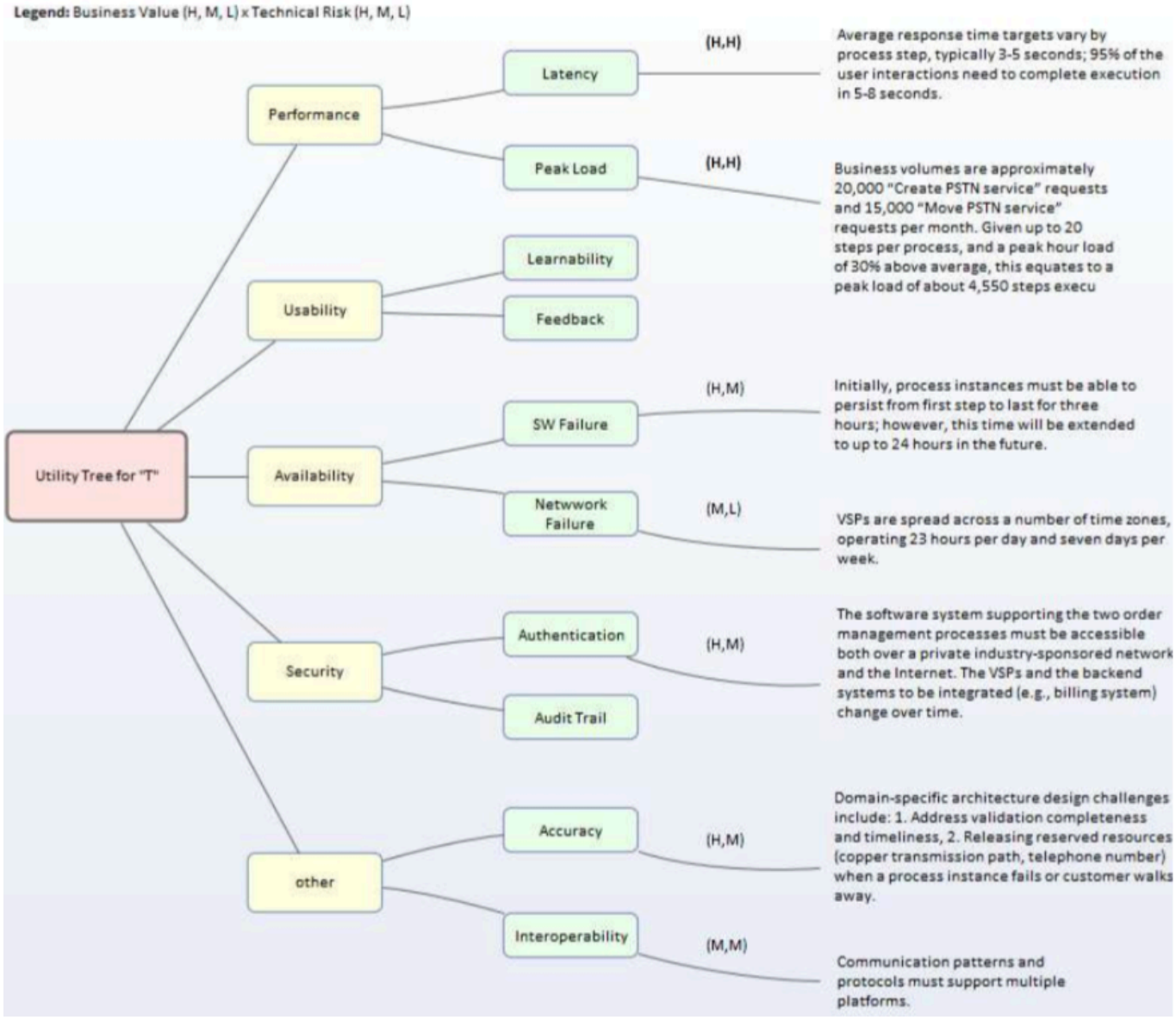
Example

Scenario for Requirement "Payment & Age-verified Purchase Availability (QAS-R1)"

Scenario Synopsis		All purchase flows that require age verification must succeed end-to-end 99.95% of the time measured monthly *(excluding scheduled maintenance windows)*. Failures of those flows must not cause inconsistent states *(no partial charge without final confirmation)* and must be recoverable within 1 hour *(automated compensation or manual operator workflow)*.
Business Goals		Age-verified purchases
Relevant Quality Attr.		Availability, Recoverability, Fault Tolerance
Scenario Components	*Stimulus*	A user attempts to complete an age-restricted purchase *(checkout)*
	Stimulus Source	External user *(gamer)* or client app invoking purchase API; age verification provider callbacks
	Environment	Production, normal operation and during marketing peaks *(see landing zones below)*. Network links to external payment and age-verification providers are considered typical *(subject to their own SLAs)*. Scheduled maintenance windows are excluded and must be announced at least 24h in advance.
	Artifact	The end-to-end purchase flow including frontend checkout module, backend Pricing service, and an Age verification service.
	Response	The system either completes the purchase with confirmation and final accounting or fails gracefully by providing an explicit error and a consistent rollback/compensation action *(no partial charges, no orphaned order records)*. Operator-visible alerts and automated rollback procedures must be triggered on failure.
	Response Measure	– **Target:** 99.95% successful end-to-end completions per calendar month; mean time to detect (MTTD) $<$ 2 minutes; mean time to recover (MTTR) $<$ 1 hour. No partial charge occurrences ($>$0 per month). – **Acceptable:** 99% success per month; MTTD $<$ 5 minutes; MTTR $<$ 4 hours. Partial charge incidents $\leq$ 1 per 10k transactions *(must be compensated within 24 hours)*. – **Unacceptable:** $<$ 98% success or any partial-charge incidents that are not compensated within 24 hours; immediate incident response escalation required.
Questions		
Issues		

1.2. QUALITY UTILITY TREES

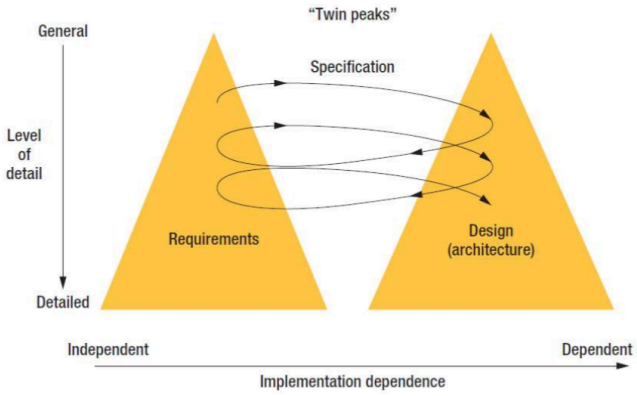
A ***tree structure*** that has a top-level taxonomy topic as its root node *(usually based on "ISO-25010 Software Product Quality" (Page 4))*. The branches then split up into finer-grained quality attributes. The ***leaves*** are QASs *(see chapter "Quality Attribute Scenarios (QAS)" (Page 7))*, prioritized by ***Business Value*** and ***technical risk*** *(Low, Medium, High or no rating if not applicable)*.



1.3. TWIN PEAKS MODEL

The architecture and requirements should be seen as ***two mountains*** that need to be worked on ***from top to bottom*** using an ***iterative*** approach. The "peaks" of the mountain are the starting points and development takes place from top to bottom in mutual interplay.

Start by formulating ***general requirements*** and start ***implementing*** them. The requirements can then be ***reworked*** in more detail depending on the experiences during implementation.



1.4. CONTEXT DIAGRAMS

1.4.1. System Context Diagram (SCD)

Represent ***systems to be built*** as black boxes. Depicts its ***interaction*** with ***external entities*** *(systems, end users)*. Identifies the ***information and control flows*** between the system and external entities.

1.4.2. C4 Model for Architecture Visualization

The ***Context, Container, Component and Classes Model for Architecture Visualization (C4)*** model consists of multiple diagrams representing each of the four C's, visualizing the architecture in ***different levels of detail***. The Class level is usually left out or automatically generated by the code. Often, supplemental ***system landscape*** *(other system the software interacts with)*, ***dynamic*** *(sequence diagram of a user story/feature)* and ***deployment diagrams*** *(where and how the software is deployed)* are added, leading to the name ***C4+3***.

Resources:

- ***Person:*** User that interacts with the system. Can be in different roles *(visitor, moderator, admin)*
- ***Software System:*** External dependencies *(external APIs, services)*
- ***Container:*** Application or data store inside the application *(Frontend, Backend, Database)*
- ***Component:*** Modules inside the containers *(APIs, Controllers, Services, logical structure within the programming language)*
- ***Relationship:*** How elements interact. Can have information on what protocol is used for communication, if any

Level 1: System Context

Shows the software system being built and how it ***fits into its environment***. This includes the ***people who use it*** and any other software systems it ***interacts*** with. It adds ***little detail*** about the system itself.

Level 2: Container Diagram

Provides an ***architecture overview***. It zooms in to the software system and shows the ***containers***, which are essentially separately deployable units that execute code or store data *(applications, data stores, microservices, etc.)*. It illustrates ***interface protocols*** and ***technology decisions*** and typically gets created during solution strategy.

Level 3: Component Diagram

Zooms into an ***individual container*** to show the components inside it. These components should map to real abstractions *(e.g., a grouping of code)* in your codebase.

Level 4: Code

Finally, if you really want or need to, you can zoom into an ***individual component*** to show how that component is implemented. There nearly is ***never the need to draw a Code diagram***, since it gives a very low level idea on how the code is structured.

2. COLLABORATIVE MODELLING

A ***domain model*** is a graphical overview over the subject area on which the application is intended to apply. It incorporates both behavior and data. The most popular domain model is the ***Unified Modelling Language (UML)***.

Motivation for Domain Modelling:

- Capture the ***essence*** of the ***problem domain***
- Establish a ***shared understanding*** *(ubiquitous language)*
- Align ***mental models*** through ***visualization***
- Align ***language*** with ***business***: Bridging technical and non-technical stakeholders
- Find the right ***abstractions*** and ***simplifications***
- Use domain knowledge as basis for ***system design*** and ***architecture***

There are often difficulties in communication between the ***Business Domain Experts*** and the ***Software Engineers*** implementing a solution in that domain. Projects within complex domains need ***shared understanding***. Knowledge is typically spread out across different persons *(business experts, developers, designers, managers…)* and stays within a certain organization. Models must serve as a ***communication tool*** between stakeholders.

Challenges of traditional modelling
Notations like UML are designed for clarity of the technical aspects, but aren't ***accessible*** for non-technical stakeholders *(like your mum)*. Discussions about them often devolve into arguing about technical details.

2.1. WHY COLLABORATIVE MODELLING?

Collaborative Modelling *(CoMo)* is a ***workshop-based technique*** for ***technical*** and ***non-technical stakeholders*** to build a ***shared understanding*** of the customer's problems and how to develop a solution for them. There is little preparation needed from the participants, the moderator of the session does most of that.

The ***benefits*** are inclusion, engagement, speed, shared language and discovery. It emphasizes people and interactions.

The two main methods used in a CoMo session are ***Domain Storytelling*** and ***Event Storming***. The moderator guides through them.

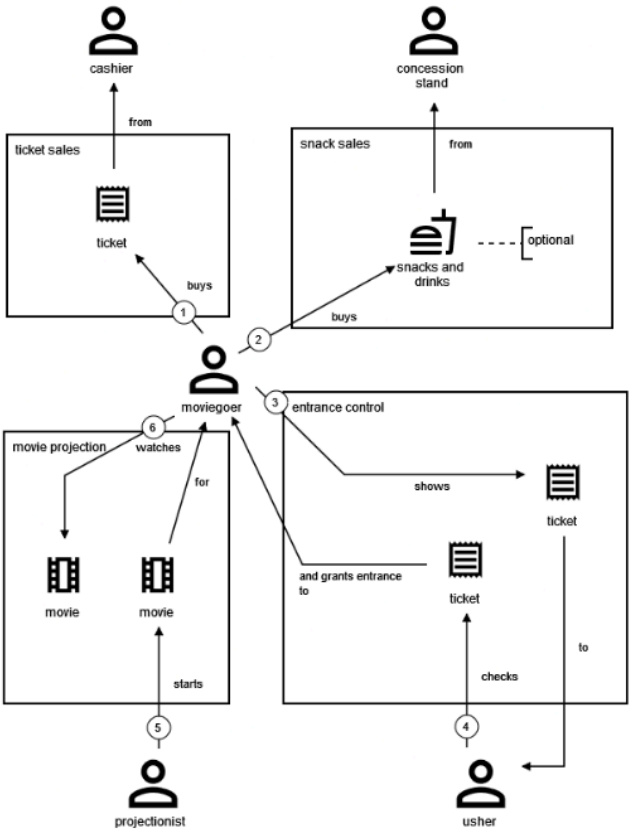
2.2. DOMAIN STORYTELLING

A diagram containing workflows within the domain, with people carrying out tasks. The goal is to ***tell a story of what is done when by whom*** with terms used by the domain experts to understand the domain and align all stakeholders.

Elements
- ***Actor:*** Person or System within the domain
- ***Work object:*** Thing an actor works at/with
- ***Activity:*** What an actor does with a work object.
- ***Sequence Number:*** Indicates in which order activities are executed.
- ***Annotations:*** Information about other cases, optional activities, possible errors or other noteworthy things
- ***Group:*** Outlines elements that belong together

The elements are always labeled with a noun/verb from the domain language. Each sequence should form a sentence in the form ***"<Actor> does <Activity> with <Work object> (with <other actor>)"***

Example
1. Moviegoer buys ticket from Cashier
2. Moviegoer buys snacks and drinks from concession stand *(optional)*
3. Moviegoer shows ticket to Usher *(Türsteher)*
4. Usher checks ticket and grants entrance to Moviegoer
5. Projectionist starts movie for Moviegoer
6. Moviegoer watches movie



Best practices
- Start modelling the default case/***happy path first***. Only then discuss what else could happen.
- Use separate work objects for each sentence, ***duplication increases readability*** in this case.
- Make work objects ***explicit***, not a part of an activity
  (i.e. instead of "Cashier —looks for available seats in→ floor plan" use "Cashier —looks for→ available seats —in→ floor plan")
- Name ***every*** actor, activity and work object
- Use ***separate icons*** for actors and work objects

2.3. EVENT STORMING

In Event Storming, the domain is ***modeled on a wall/white board*** with ***differently colored post-its*** on a ***timeline***. The whole session is guided by a moderator.

Common Concepts across all types (Color = Color of the Post-it)

- ***Domain Events:*** An event that happened in the past and the business cares about *(e.g. "Item added to cart")*.
- ***Hot Spot:*** Things the stakeholders do not agree on or are unclear.
- ***Actor/Agent:*** (Group of) people involved in a domain event.
- ***System:*** IT system used as a solution for a problem. Wide Post-it.
- ***Command/Action:*** Decision, actions or intent, either automated or manual

The general procedure of a Event Storming workshop is:

1. Storm out the business process by creating a series of Domain Events on sticky notes.
2. Create Commands that cause each Domain Event.
3. Associate the Entity/Aggregate on which the Command is executed and that produces the Domain Event outcome.
4. Draw boundaries and lines with arrows to show flow on your modeling surface.
5. Identify the various views that your users will need to carry out their actions, and important roles for various users

The original definition splits Event Storming into three types. Note that in the exercises we didn't make this distinction.

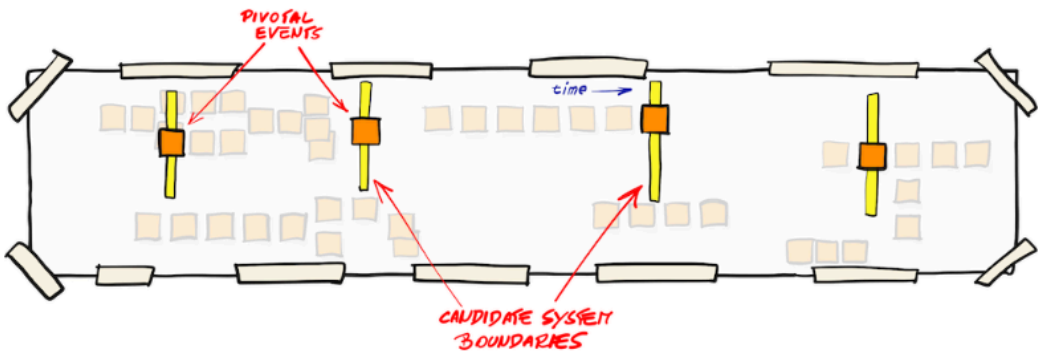
Big Picture (less detailed)	***Process Modelling***	***Software Design*** (more detailed)
– Create a shared state of mind – Explore business/domain model – Identify Bounded Contexts **Used for:** Broad organizational storytelling	– Assess specific process – Find bottlenecks and identify parts of the system to decouple from existing software **Used for:** Detailed workflows	– Design clean and maintainable event-driven software – Derive shared language and domain model within a bounded context **Used for:** Implementation conversations

Best Practices

- ***Clarify the purpose:*** The moderator should explain to the participants why the session is done
- ***Start simple:*** Get the basic steps of the process on the timeline first without too much thought on where they belong on the timeline
- ***Model different stories:*** Model different types of scenarios *(e.g. for a game store, model the flow for developers and gamers)*
- ***Match scale to scope:*** Choose the appropriate event storming type
- ***Experiment and iterate:*** Move sticky notes around, there is no need to be correct or perfect at first
- ***Lean on facilitation, not just technique:*** The moderator should clearly guide and split the session into different steps
- ***Blend styles if needed:*** For some context, the group can switch to a different event storming type

2.3.1. Pivotal Events & Emerging Bounded Context

The most significant domain events get turned into ***pivotal events***. *(e.g. for a online shop: "Article Added to Catalogue", "Order Placed", "Order Shipped", "Payment Received" and "Order Delivered")*. These are then placed on a long yellow tape called ***Candidate system boundaries*** that separate different steps in the product.

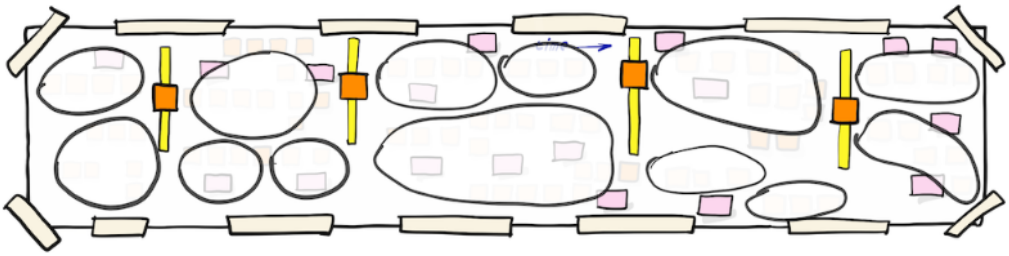


Pivotal Events are a great source of information.

Criteria for a pivotal event

1. ***Events that trigger significant downstream activity:*** The "So What?" test *(Event causes lots of changes)*.
2. ***Events that represent key business decisions or policy enforcement:*** Decision points and policy application.
3. ***Events that involve hand-overs to external parties:*** Boundary crossings and external triggers.
4. ***Events that lead to lasting state changes:*** Significant data updates and process milestones.
5. ***Events that indicate potential bottlenecks or failure points:*** Points of contention and exception handling.
6. ***Compliance and regulatory significance:*** Events tied to legal or regulatory requirements.
7. ***Change in resource allocation:*** Events causing significant shifts in personnel, time, or budget.
8. ***Impact on multiple stakeholders:*** Events affecting various departments or stakeholders.
9. ***High business value realization potential:*** Events that correlate with achieving key business objectives.

Within the candidate system boundaries, different post-its can be grouped together into ***emerging bounded contexts***. See "Bounded Context" (Page 14).



Emerging bounded contexts after a Big Picture EventStorming.

2.3.2. Big Picture Event Storming

Goal: Assess health of existing business or explore viability of new business.

Additional Concepts:

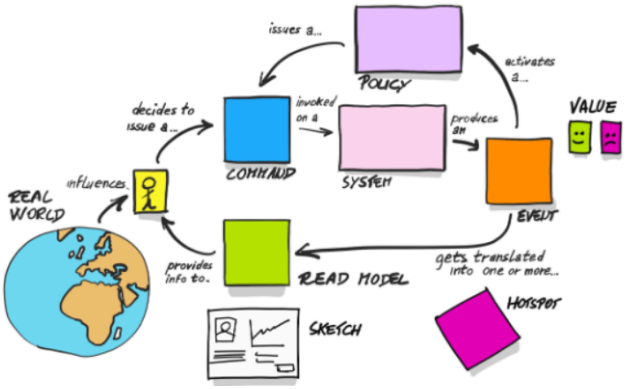
– ***Opportunity:*** Possible improvements to a hot spot
– ***Negative Value***/***Positive Value:*** Apply after timeline has been made consistent

2.3.3. Process Modelling Event Storming

Goal: Assess the health of a specific process in the company.

Additional Concepts

– ***Policy:*** "Whenever X happens, we do Y", automated or manual process. In between a Domain Event and a command/action.
– ***Query Model/Information:*** Information needed by an actor to make decisions. Also includes how information is presented to the actor, through UI/Forms.



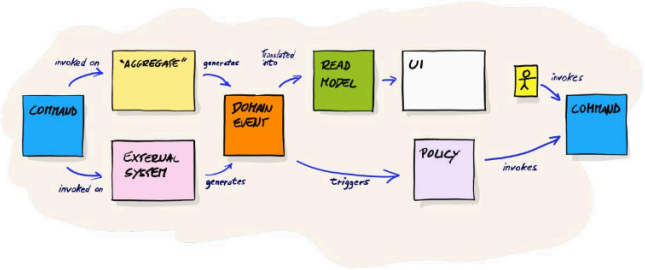
2.3.4. Design Level Event Storming

Goal: Design maintainable event-driven software

Additional Concept

Constraint: Previously called "aggregate". Depending on the Event Storming Style used it can mean:

– ***Objects of data*** on which commands are executed *i.e. the actual data the application interacts with*
– A ***restriction*** that has to be accounted for before a command/action can be performed. The basis of business rules.



Note: *In the exercise session, we didn't differentiate and did a mixture of all three event storming types. It sits somewhere between Big Picture and Design Level Event Storming.*

2.4. COMPARISON OF COMO PRACTICES

Practice	*Usage*	*Strengths*	*Weaknesses*	*Space*
Big Picture Event Storming	Modelling/designing a enterprise, business or domain	Adaptable and quick to learn, Chaotic nature gives a lot of insight	A lot of people in one room, requires experience of moderator, only works with a timeline	Problem space
Process Modeling Event Storming	Modelling/designing a story, process or timeline		Difficult concepts to grasp, can feel like a high time investment, only works with a timeline	Problem and solution space
Design Level Event Storming	Designing software for stakeholders needs			Solution space
Domain Storytelling	Modelling one specific scenario, process or timeline	No learning curve, instant documentation	Structured approach to lower amount of discovery	Problem and solution space

When should you choose…

Both focus on collaboration between domain experts and finding bounded contexts. Can also be combined.

Event Storming	*Domain Storytelling*
– Discover existing structure – Storm new ideas, be creative – Model complex domains, where a story can't yet be put into words – Model processes with a strong time reference – Scales better with many people – Participants explore domain, less moderating Displays results on a timeline: **What happens when?**	– Documentation is required *(use a modelling tool to have something "docs-ready")* – Collaboration/communication between actors must be modelled – Workshop needs to be recorded *(i.e. remote meeting)* – Company culture prefers structured approach – Moderator channels inputs and does the modelling Displays collaboration between actors: **Who does what with whom?**

3. STRATEGIC DDD, SOLUTION STRATEGY & ARCHITECTURAL DECISIONS

The requirements for the ***development process*** often differ from NFRs. A ***loosely coupled architecture and organization structure*** is key to fulfilling them.

- Release often, iteratively & quickly *(continuos delivery)*
- Release features fast *(time to market)*
- Update only part of a system *(each system can deploy separately)*
- Scalable organization *(split into multiple teams)*
- Being able to respond to changes fast

3.1. STRATEGIC DOMAIN-DRIVEN DESIGN (DDD)

With ***Strategic DDD***, a high-level overview over the business domain can be created and different parts can be bundled within ***bounded contexts***. These are then placed in a ***context map*** that showcases the relationships and integrations of different bounded contexts, see Chapter "Context Mapping" (Page 16). The process is often done iteratively during the development of a project. ***Tactical DDD*** focuses on the design of the model inside a bounded context, see "Tactical Domain-Driven Design" (Page 29).

3.1.1. Subdomains

Subdomains represent ***the problems your customers have*** – the ***problem space***. They can be split into three types:

Core Domain	***Supporting Domain***	***Generic Subdomain***
Represent the core concept of your business domain – these are ***your selling points***. Highly domain-specific knowledge. Mostly or fully created in-house.	Parts that support your business, but don't belong to your core competencies. Usually requires ***some amount*** of domain knowledge. Use existing components with tweaks.	Needed parts, but don't capture or communicate core business knowledge. Little to ***no domain knowledge*** is needed. Usually off-the-shelf solutions are used.
Example: A dentist has three problems:		
Fixing patient's teeth The main work people pay for. Improvements in this area will win new customers.	**Making appointments for the patients** The appointment logic must follow domain specific rules *(Appointment length depends on the type of procedure, availability of equipment and personnel, emergencies…)* Improving here may win new customers, but it is not the main thing they visit for.	**Billing** *(Invoices, payments, taxes, …)* Governed largely by external rules. Improvements in this area don't win new customers. Custom building a billing system would add little strategic value

3.1.2. Bounded Context

DDD suggests to decompose systems into ***Bounded Contexts***. They represent the ***solution space*** and should implement parts of one or multiple subdomains.

Bounded contexts establish ***boundaries*** between domain models *(e.g. the Sales Context vs. Support Context)*. The concepts within a Bounded Context have distinctive meanings in order to establish linguistic boundaries. Words can have different meanings in different contexts. Usually, ***one bounded context per team*** is created.

Why bother defining bounded contexts?

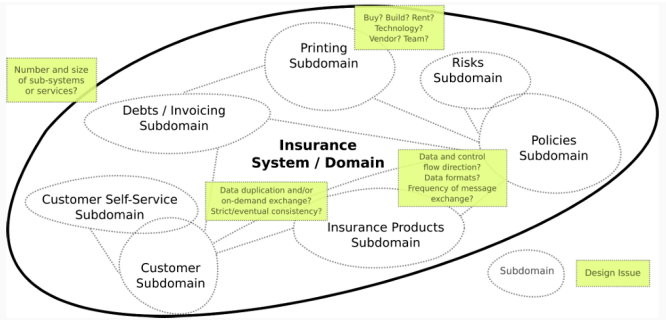
- Same term, different meaning *(homonym)*
- Same concept, different use *(polyseme)*
- External system differences *(heterogeneity)*
- Scaling up the organization *(multiple teams)*

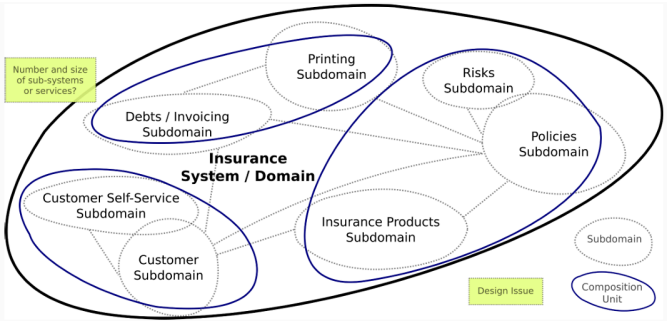
3.1.3. Subdomain Diagram

To begin with DDD, a diagram containing all relevant subdomains is created. This type of diagram doesn't have an official name, but to avoid confusion it will be called ***"Subdomain Diagram"*** in this document.

It shows the different subdomains of the client. Domain experts assist in drawing the basic relationships between the subdomains. If any questions arise at this stage, a ***design issue*** can be created. They address high level topics like

- Do we ***implement*** this subdomain ***ourselves*** or do we use a third-party-product?
- ***Technologies*** that could be used for implementation
- Basic ***architectural questions*** *(new module, (multiple) microservices?)*
- ***Integration*** into ***existing systems*** necessary?



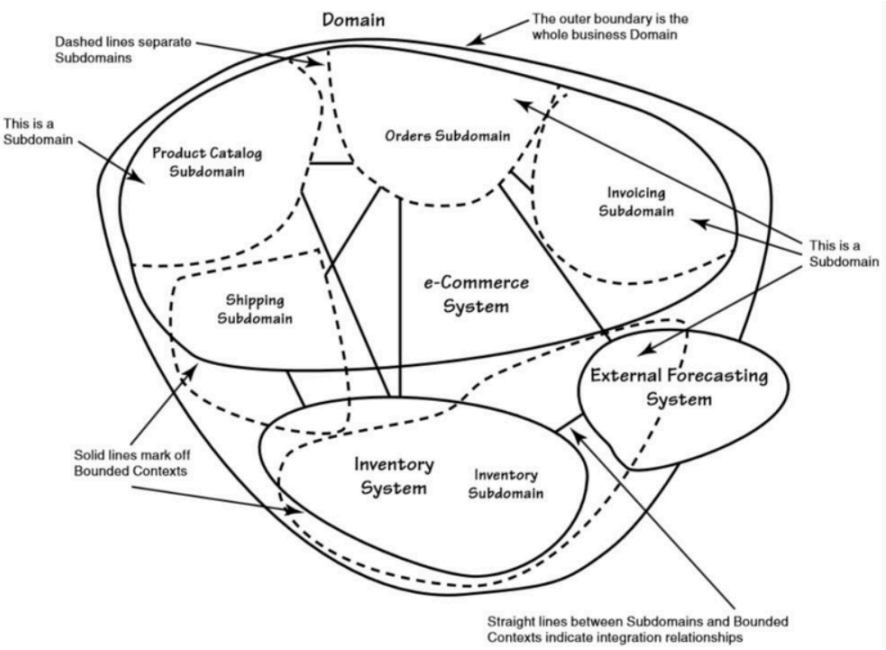


The subdomains can be grouped into preliminary ***bounded contexts*** *(blue in the second image)*. "Bounded Context" can be used interchangeably with "Composition Unit" in this stage.

Overview of Subdomains & Bounded Contexts

The diagram showcases ***subdomains*** and ***bounded context*** within a domain. It is not part of the DDD process, it just provides an ***overview*** over both concepts. Relationships between subdomains and bounded contexts represent a "x implements y"-relationship.

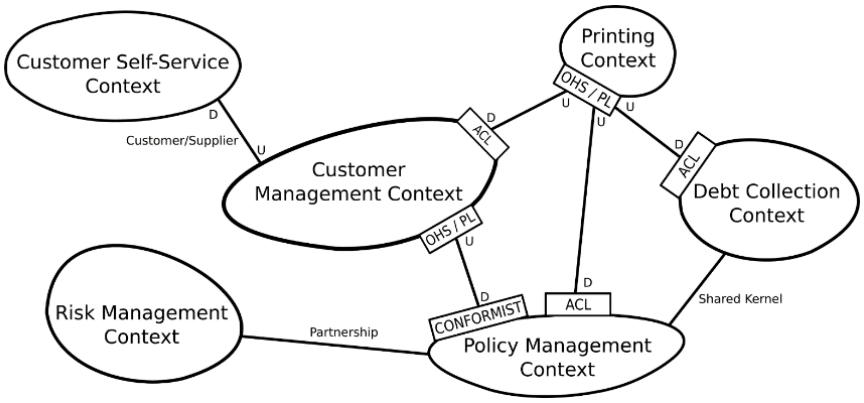
Bounded Contexts don't have to map 1:1 to a Subdomain and a Bounded context can overlap with multiple Subdomains.



Subdomains	***Bounded Contexts***
Problem Space *(Problems the customer wants solved)* Result of Object-oriented analysis *(OAA)*	Solution Space *(Solutions that address these problems)* Result of Object-oriented Design *(OOD)*
Grouped by their business importance *(Core, supporting, generic subdomain)*	Grouped by their relationships with each other *(Symmetric, asymmetric, upstream integration, downstream integration)*
Should always be *(partly)* covered by at least one bounded context	Can implement one or multiple subdomains partly or fully

3.2. CONTEXT MAPPING

DDD produces a ***Context Map*** that defines ***how*** bounded contexts ***integrate*** – it illustrates the information flow. We take the bounded contexts we discovered in the subdomain diagram and ***label the relationships*** between the bounded contexts. A context map only showcases the design of the solution, so there are no subdomains, only bounded contexts.



3.2.1. Relationships

Each relationship in a context map is either ***symmetrical*** or ***asymmetrical***. Each Upstream-Downstream ***relationship*** usually has a Upstream/Downstream integration ***pattern*** on each end.

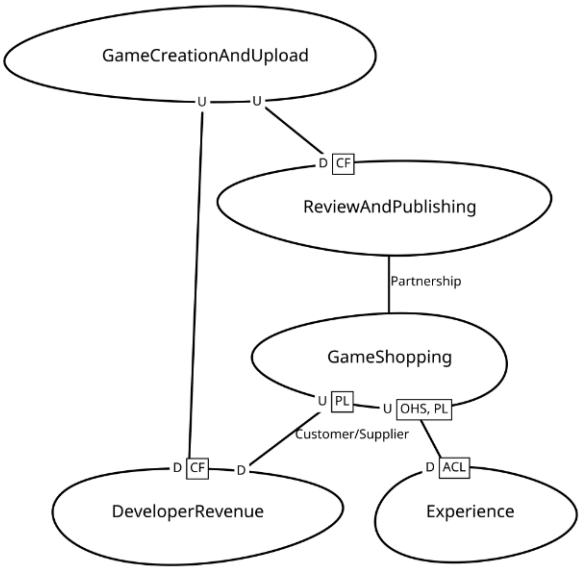
	Type	***Description***
Symmetric Relationships	Shared Kernel *(SK)*	Two bounded contexts share a part of their domain models. The shared part is typically realized as a library maintained by both teams.
	Partnership *(P)*	Two teams cooperate together and can only succeed or fail together *(delivery failure of one leads to delivery failure of both)*. These teams may share CI/CD infrastructure and always release together.
Asymmetric Relationships	Upstream – Downstream *(U → D)*	The actions of the upstream group affect the downstream group, but not vice versa. The upstream context exposes parts of its domain model to the downstream context. Each U-D-Relationship also has one U/D integration pattern on each end.
	Customer – Supplier *(S → C)*	Upstream-downstream relationship where the Customer *(downstream)* can highly influence the Supplier *(upstream)*. The supplier respects customer's requirements and plans accordingly.
Upstream Integration Pattern	Open Host Service *(OHS)*	Upstream context which provides a uniform API to multiple downstreams. If they have mostly the same requirements, implement a single API instead of integrating with each individually.
	Published Language *(PL)*	Upstream context defines a common language used to translate between models – e.g. JSON. Often combined with OHS.
Downstream Integration Pattern	Anticorruption Layer *(ACL)*	Downstream context translates between domain models to protect its own model from upstream changes – a ***Isolation-/Translation-Layer***. The opposite of Conformist.
	Conformist *(CF)*	Downstream decides to conform to the upstream model – its changes have direct impact on downstream. Opposite of ACL.

Example Context Map for Fair Game 3002:
The bounded contexts in this scenario are:

- ***Game Creation and Upload:*** This part of the system shall allow developers to create games, upload them, and later publish new releases/versions of the games.
- ***Review and Publishing:*** This context is all about the game review, compliance checks, pricing, and publishing of a game.
- ***Game Shopping:*** This context is the online shop for the gamers. It covers the game catalog (searching, filtering) as well as buying and downloading games.
- ***Experience:*** This part of the system is all about the experience for the gamers and developers. It covers ratings, reviews, etc.
- ***Revenue:*** This part of the system must receive all kinds of information about games that have been sold; so that it can payout the developers via bank transfer.
- ***User Profiles:*** Manages the user profiles for game developers as well as gamers.

Solution:

- The ***UserProfile*** context has been omitted as many contexts will potentially need profile information about the gamers and game developers → Cross-cutting
- The ***GameCreationAndUpload*** *(uploaded games)* will provide information that will be needed ***ReviewAndPublishing*** *(the reviewers and the publication process)*.
- The ***GameShopping*** context will need information of the ***ReviewAndPublishing*** context to add accepted games to the catalog.
- The ***DeveloperRevenue*** context will need information about the UserProfiles *(Developers)*, ***GameShopping*** *(the sold games)*, as well as ***GameCreationAndUpload*** *(game information such as the price; maybe that could also be provided by the GameShopping context)*.

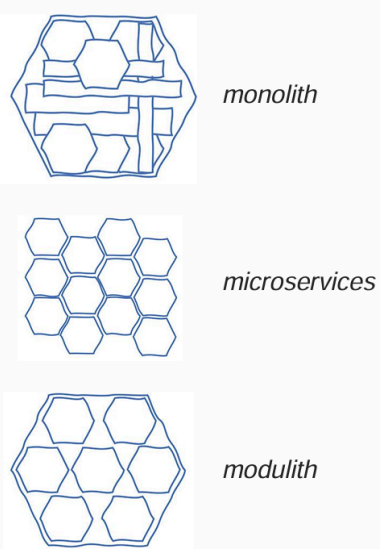


3.3. SERVICE DECOMPOSITION

Service Decomposition is the umbrella term for architectural styles that allow us to ***decompose*** a software system into ***subsystems***, (micro-)services and modules. Two main ways are ***Service oriented architectural style (SOA)*** *(serves as the basis for microservices)* and ***Moduliths***.

3.3.1. Monolith, Modulith and Microservices

One of the biggest architectural decisions is whether to build a ***monolith*** or ***split*** the software up into ***independent microservices***. However, starting your project as a microservice increases the ***complexity*** significantly without any immediate benefits, thus violating the ***You ain't gonna need it (YAGNI) principle***. Additionally, microservices only work well if the ***boundaries*** between the services are ***stable*** – which is highly unlikely when a project is just starting up. A ***monolith*** allows you to explore the boundaries of the components and easily move stuff around. However, without proper planning, it will likely lead to some unmaintainable behemoth.



A way to combine these two architectures would be to start building a ***Modulith***: The components are split up into ***different modules*** like in a microservice architecture, but they are always ***deployed as a whole***. This keeps the code base "under one roof" and allows to see dependencies relatively quickly. Strong couplings between components can be spotted easier and reworked. If done correctly, a component should be able to be broken off the modulith without affection other components.

3.4. SOLUTION STRATEGY

The solution strategy is about the fundamental decisions that shape a system's architecture. These are the ***big decisions*** made during the in the beginning of the project – i.e. the ***Inception & Elaboration Phases*** in Rational Unified Process (RUP).

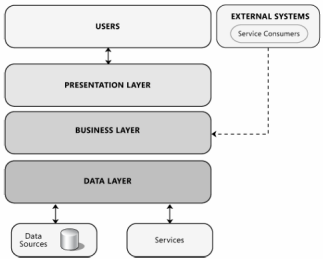
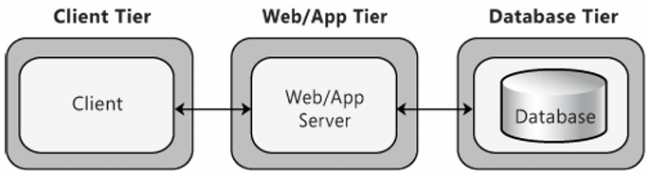
1. ***Technology decisions:*** Programming languages, Database type etc.
2. ***Top-level decomposition:*** Architectural patterns, design patterns
3. ***How to achieve key quality goals:*** Formulating (Non-)Functional Requirements
4. ***Relevant organizational decisions:*** Select a development process, delegating tasks to third parties…

What makes architectural decisions (ADs) big?

1. ***High architectural significance score***: High score in ASR test *(see chapter "Architecturally significant requirement (ASRs)" (Page 2))*, a "H/H" in QAS tree rankings *(see "Quality Attribute Scenarios (QAS)" (Page 7))*
2. ***High financial investment and/or tough consequences:*** Software licenses, training, consultants, cloud operations, people impact…
3. ***Long time to execute:*** Need for PoCs, training, recruiting…
4. ***Many or still unclear outgoing dependencies:*** "One thing leads to another" – other ADs depend on it
5. ***Take a long time to pass Definition of Done:*** Many stakeholders, goal conflicts, hard to revise…
6. ***High level of abstraction:*** Architectural style, composite patterns…
7. ***Problem/solution space outside of team's comfort zone***

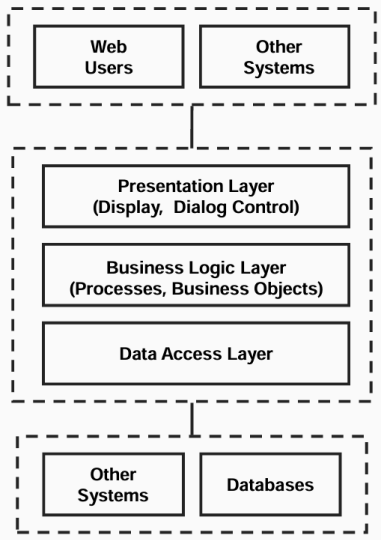
3.4.1. Layers and Tiers

An application can be split into layers and tiers. Each layer/tier only talks to its neighbors.

Layer	***Tier***
Separate concerns *(Presentation layer, business layer, data layer)*	Distribute workflow *(Client tier, Web app tier, Database tier)*
Top-to-bottom flow	Left-to-right flow
	

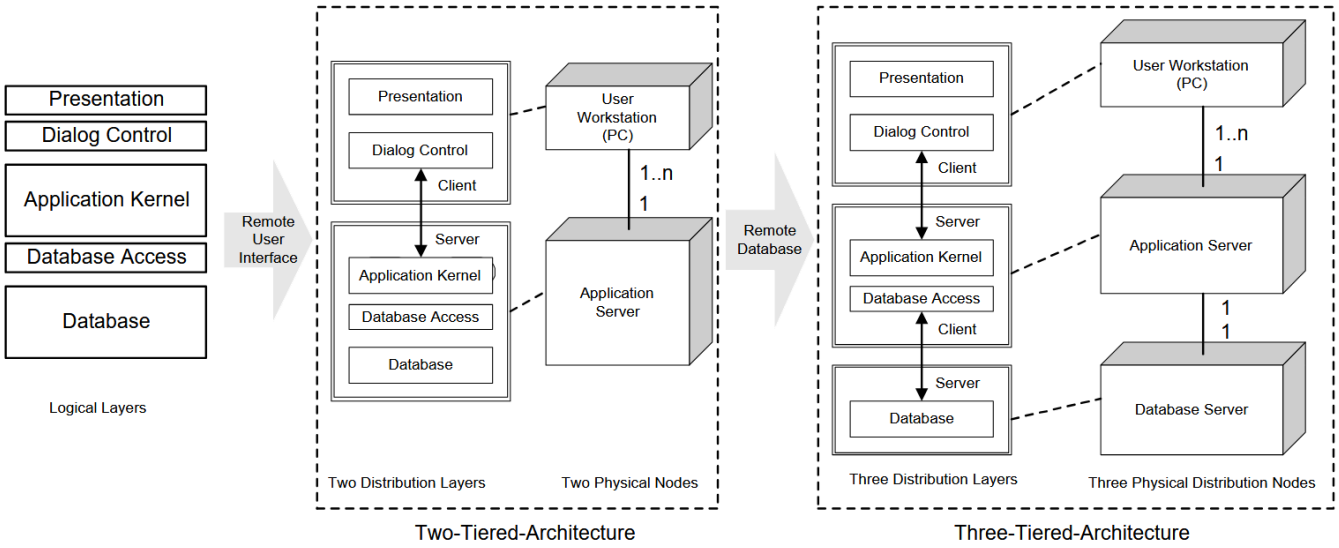
Tiers are essentially just ***layers applied twice***: in the ***logical*** and ***physical*** view. With tiers, we can define process/server boundaries *(dashed lines)*.

- ***End users and external systems*** only talk to the presentation layer to isolate them from the rest of the backend
- ***Presentation layer*** talks to the business logic to support multiple presentations of the same logic
- ***Business logic*** uses the database access layer to communicate with the database and other backend systems. The data access layer can be swapped in or out if a change in the other systems occurs



Partitioning the application into server/client components with well-defined boundaries is known as ***Distribution patterns*** or ***Client/Server Cuts (CSCs)***.

This separation allows us to move from a single logical layer that may or may not reside on the same machine into a ***Two-Tiered-Architecture*** *(Remote User Interface: Client-Server)* and a ***Three-Tiered-Architecture*** *(Remote Database: Client-Server-Database)*



3.4.2. C4 and arc42 in Architecture Overview

The ***C4 Container diagram*** provides an architectural overview ***similar to Tiers***. Both describe layers, modules or components of the system that are run and deployed independently. The container diagram is typically created during solution strategy and refined iteratively and incrementally later as needed. It illustrates the Client-Server Cuts and other big decisions like the interface protocols and implementation technologies.

The ***arc42 model*** goes one step further: It provides a ***whole template*** for documenting your solution strategy and how to prioritize.

3.5. DOCUMENTING ARCHITECTURAL DECISIONS (AD)

Architectural Decisions (AD) are design decisions that are costly to change. They should therefore be well-documented in ***Architecture Decision Records (ADR)*** in order to:

- Make the rationale and ***justification*** of ADs ***explicit***
- Avoid ***unnecessary reconsideration*** of the same issues
- ***Preserve design integrity*** in function components
- Provide a ***single place*** to find important decisions
- ***Reference*** of documented ***decisions*** for new people
- Ensure that the architecture is ***extensible*** and ***evolvable***

ADRs can also be embedded into code *(comments, custom annotations)* or formatted with markdown and versioned. Templates like ***Markdown Architectural Decision Records (MADR)*** exist.

3.5.1. The Y-Template

The Y-Template is a ***sentence structure*** that allows efficient notation of ADs. It emphasizes tradeoffs between qualities *(requirement vs commitment, pros vs. cons)*

In the context of <use case uc *and/or component* co*>…* *…facing <non-functional concern* c*>,*

…we decided for <option o_1*>* **Y** *and neglected <options* o_2 *to* o_n*>,*

…to achieve <quality q*>*

…accepting downside <consequence c*>.*

Example 1: Combine Messaging and Remote Procedure

In the context of the order management scenario at Evil Inc.,
… ***facing*** the need to process customer orders synchronously without losing any messages,
… ***we decided to*** apply the messaging pattern and the RPC pattern
… ***and neglected*** File Transfer, Shared Database and physical distribution *(local calls)*
… ***to achieve*** guaranteed delivery and request buffering when dealing with unreliable data sources,
… ***accepting that*** follow-on detailed design work has to be performed and that we need to select, install and configure a message-oriented middleware provider.

Example 2:
In the context of the Fair Game 3002 platform,
… ***facing*** the need to handle high load and availability demands on shopping and payment flows *(as captured in SMART NFRs such as 99.95% purchase availability and fast response under peak loads)*,
… ***we decided to*** separate the GameShopping Bounded Context (BC) from the rest of the system into its own backend tier/microservice
… ***and neglected*** a fully monolithic deployment of all bounded contexts together
… ***to achieve*** higher availability, independent scalability, and fault isolation for the shopping flows,
… ***accepting that*** this comes with increased deployment and operational complexity, more inter-service communication, and potentially higher infrastructure cost.

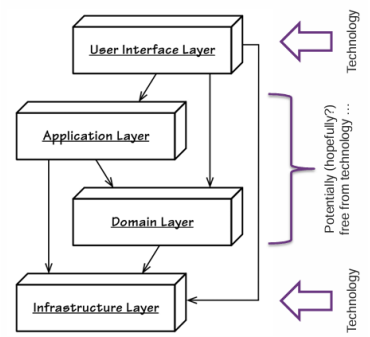
4. RINGS, IOC, ARCHITECTURAL VIEWPOINTS

One issue with the layer architecture is that technology/infrastructure exists at the top and bottom:

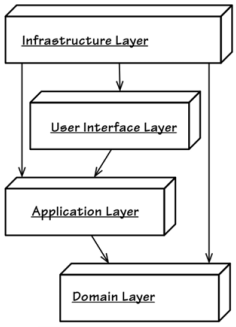
– ***Top:*** User interface, APIs…
– ***Bottom:*** Databases, file system…

Therefore, it is harder to keep the business logic ***clean and independent*** of technology and infrastructure dependencies. During ***tactical DDD***, we usually want to avoid dependencies from domain to infrastructure → Separation of concerns, technology vs. domain/business logic.
The ***Dependency Inversion Principle (DIP)*** helps us to achieve this with layers as well through ***interfaces***.

Original Layer architecture



Layers with DIP (Interfaces)



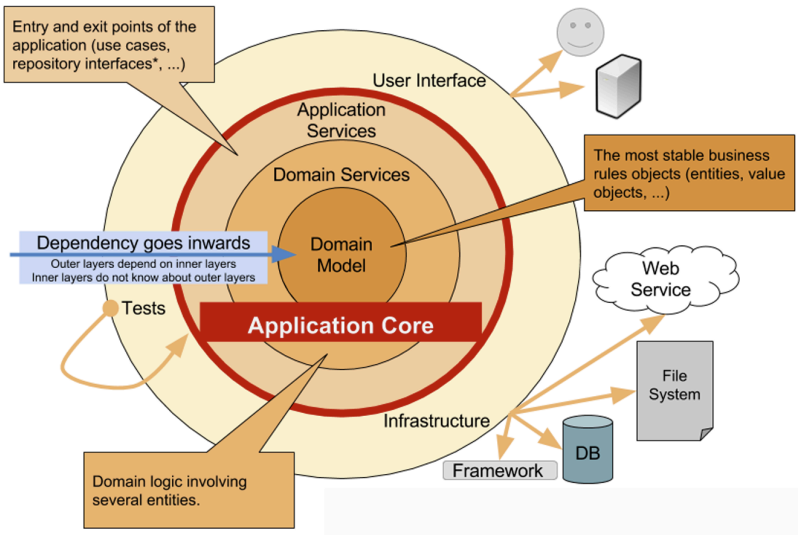
4.1. ARCHITECTURES WITH RINGS

A ***ring*** in software architecture is a ***zone of responsibility*** within a software. Dependencies are only allowed to point to inner rings, but never outer ones. ***Inner rings*** represent the essential, business-focused and stable, while ***outer rings*** represent variable, infrastructural and replaceable things.

4.1.1. Onion Architecture

An alternative approach to layers is the ***Onion Architecture***. The outer layers depend on the inner layers, but not vice-versa. Inner layers define interfaces, outer ones implement them.

– ***Domain model*** consists of business entities and core rules
– ***Domain services*** interact with and change the objects of the domain model.
– ***Application Services*** interact with external systems to persist objects, receive data from external systems
– ***Outside*** of the application core are things that change often: Connection to external systems and databases, user interfaces and integration tests.



Comparison with Clean Architecture

The onion architecture is similar to Clean Architecture by Robert C. Martin *(aka Uncle Bob)*. Both feature inward dependencies and have similar elements in their respective layers. There are some differences, however.

	Onion Architecture	*Clean Architecture*
Layering and Dependencies	Enforces strict layering, where each layer depends only on the layer directly inside it	More flexible with layering; you can skip layers and still keep the core logic independent
Domain Focus	Very domain-driven. All about keeping the domain model central.	Focuses more on framework independence and flexibility with use cases driving the logic
Use Cases and Adaptability	Focuses on the domain and business rules	Introduces uses cases or interactors to handle application-specific rules, making it adaptable to changing system needs.

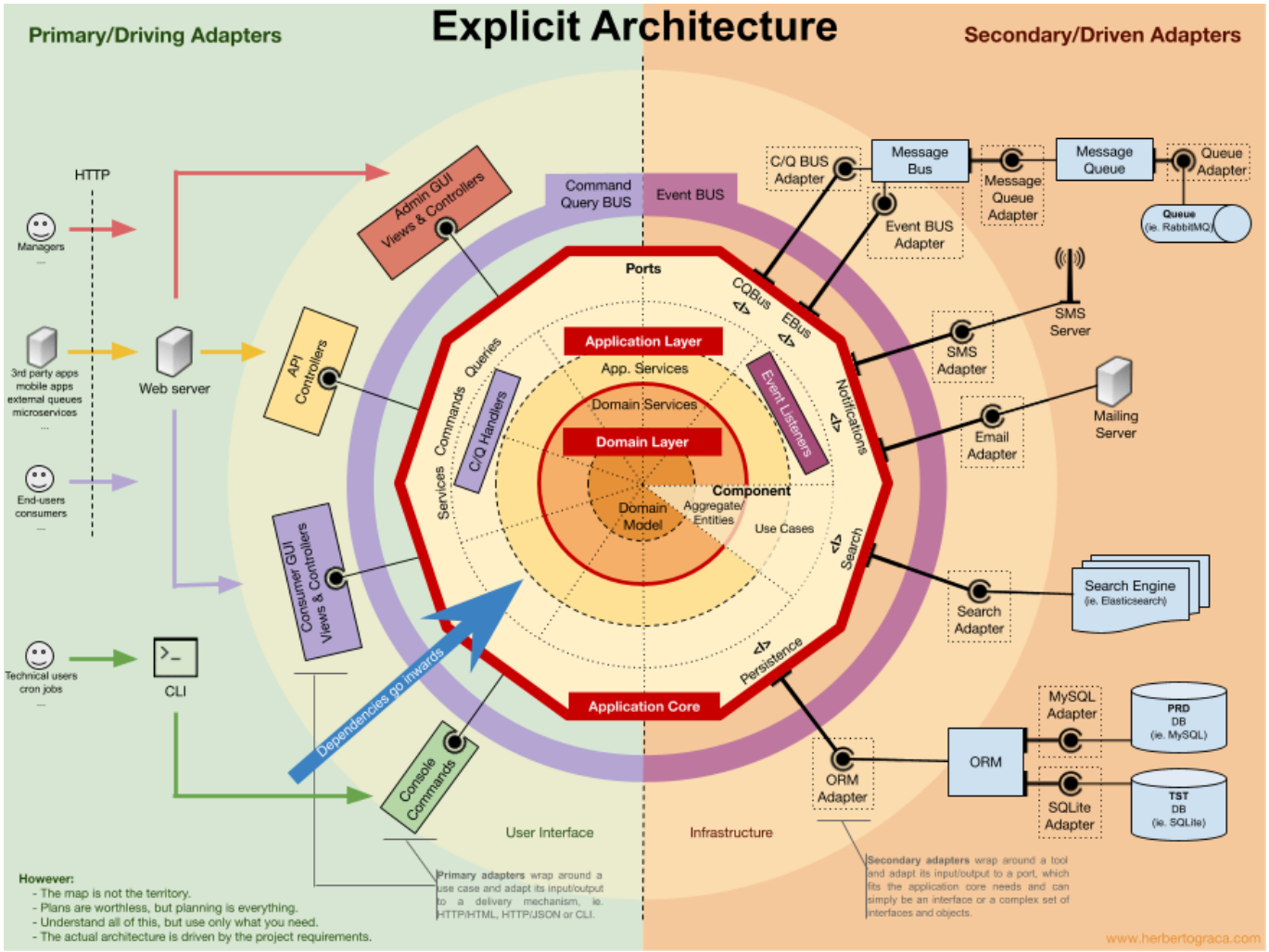
4.1.2. Hexagonal Architecture/Ports and Adapters

Hexagonal Architecture or Ports and Adapters expands Onion Architecture by dividing the outer layer into two parts:

- ***User Interface (left/green):*** Everything that turns information into any kind of user interface
- ***Infrastructure (right/orange):*** Code that connects the application core to tools like a DB or 3rd party API

For ***interaction*** between the application core and the outer layer, hexagonal architecture proposes ***two items***:

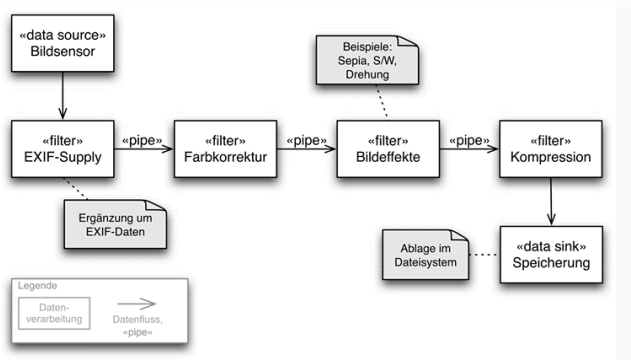
- ***Ports:*** Specification of how a tool can use the core or how it is used by the core. Usually a single interface, but it can also be multiple interfaces and data transfer object (DTOs).
- ***Adapters:*** Wrap around a Port and use it to tell the application core what to do



4.2. IMPORTANT ARCHITECTURAL STYLES & PATTERNS

4.2.1. Pipes and Filters

Simple architecture to ***transfer data*** and ***apply filtering logic***.



When to use:

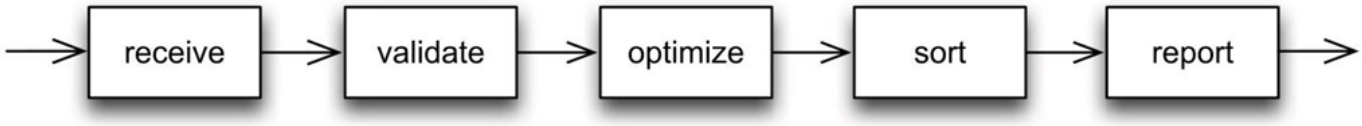
- Results are produced through a ***chain*** of processing steps
- Individual data records can be processed ***independently*** of each other
- Data format on the pipes remains ***stable***

Advantages

- ***Loose coupling*** and ***low dependencies*** between filters
- Filters can be easily ***replaced*** with alternative implementations

4.2.2. Batch Pattern

Processes input data through a ***sequence of transformations***, usually strictly ***sequential***. Data is sent in "batches" from one processing step to the next. Still widely used in ***offline data processing*** *(banks, insurance)*, now often enhanced with DBs, error handling, and orchestration.



When to use:

- Results produced through a ***fixed chain*** of ***processing steps***.
- Intermediate results can be stored in a ***common format*** *(e.g. files)*.
- Steps may ***depend*** on multiple or all input data *(aggregations, cross-references)*.

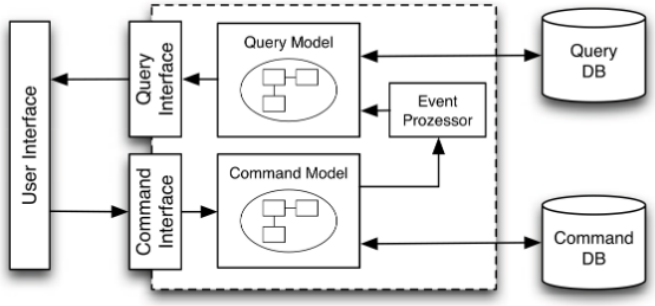
Advantages:

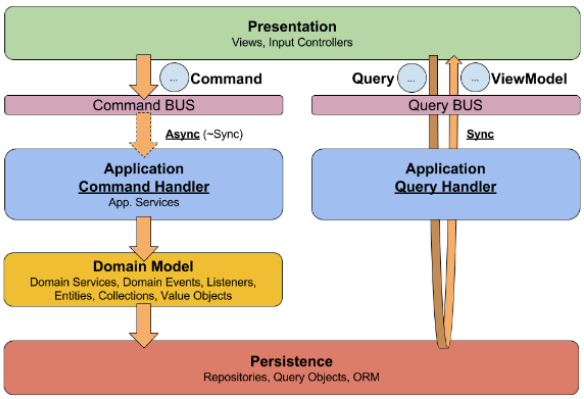
- Simple, functional ***decomposition*** of the system.
- Interfaces are usually ***straightforward*** *(e.g., input/output files)*.

4.2.3. Command Query Responsibility Segregation (CQRS)

Separates query *(read)* and ***write*** *(command)* responsibilities at the system level. Enables ***parallelism*** and ***scalability*** – queries can run independently and be optimized for performance. ***Commands trigger events*** that update the query side *(via messaging or queues)*. Commands and queries can used different models or databases – CQRS supports ***eventual consistency***. Fits scenarios with ***few complex writes, but many frequents reads***.

Trade-off: Higher complexity and data synchronization effort.





When applying tactic DDD, only the command side will use the domain model/aggregate. Queries are usually handled with their own "query model"/"read model".

4.2.4. Event-driven architectures

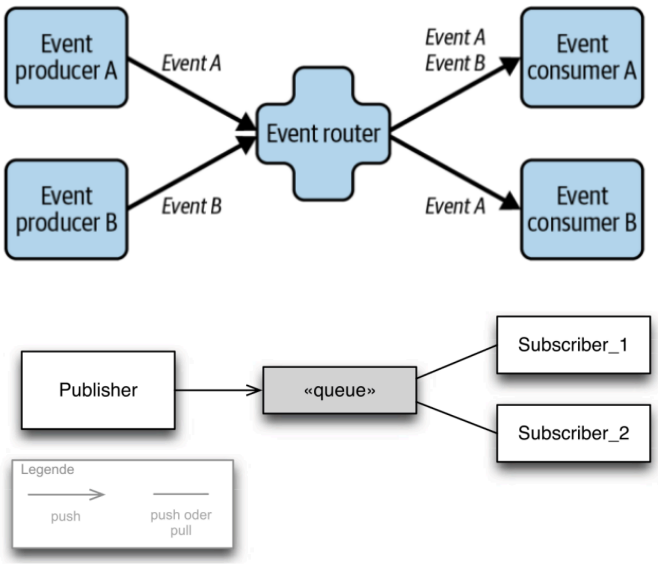
The system consists of ***Event Sources*** and ***Event Sinks***, communicating via events, not direct calls. Enables ***loose coupling*** and ***asynchronous communication*** *(implicit invocation)*. Offers ***high decoupling*** and ***scalability***, but ***increased complexity*** and ***operational overhead***.

Common variants:

- ***Publish-Subscribe:*** Sources broadcast events, subscribers react selectively
- ***Message/Event Queues:*** Events buffered and delivered asynchronously *(fire-and-forget)*

When to use:

- ***No immediate response*** needed
- ***Integration*** of heterogeneous systems
- Producers send messages ***faster*** than consumers can process them.



4.3. ARCHITECTURAL BEST PRACTICES

- Use ***existing architectures*** as guidance, not dogma
- ***Adapt patterns*** to fit your team, domain and constraints
- ***Combine elements*** that solve your specific problem
- ***Avoid over-engineering*** for purity's sake
- The best architecture is the ***one that works for you***
- **But:** Stay disciplined – ***don't break your own architectural boundaries*** or introduce unwanted dependencies

It is best to accept that architecture usually evolves iteratively:

- Requirements, constraints, and influencing factors ***change over time***.
- The ***final solution*** often ***differs*** from the ***initial concept***: development targets move continuously *("moving targets")*.
- ***Iterative*** and incremental ***development*** helps solutions and goals ***converge*** over successive iterations.
- Software architectures typically ***evolve*** through cycles and iterations.
- ***Design decisions*** and their implementation can ***influence*** organizational processes and trigger new requirements.

To ensure your rings, layers or modules are ***actually enforced***, use ***Architecturally Evident Code*** with tools like ArchUnit that creates unit tests that check whether your code actually respects your architecture.

4.4. INVERSION OF CONTROL (IOC)

Inversion of Control, or the ***Hollywood Principle***, is the idea that the framework, not the application controls the program flow: **"Don't call us, we call you"**. The code is called vie ***callbacks***, ***hooks*** and ***event handlers*** of the framework. You insert behavior into a framework and it orchestrates sequencing and calls. This distinguishes it from a library, where you call the code. ***Dependency Injection*** is a part of Inversion of Control.

4.4.1. What is a framework?

A framework promotes ***consistent approaches to solving problems***. It applies rules and policies to…

- ***reduce*** tedious and error prone programming ***work*** → **better code**
- ***raise the level of abstraction*** and convenience by viewpoint / by layer → **less code**

A framework should be considered in the following places:

- ***Routine work*** that has to be done in many places *(not domain-specific, e.g. logging, testing, configuration…)*
- ***Risky activities*** *(e.g. embedded system development facing real-time requirements like guaranteed response times)*
- Need for ***Dependency Injection***, to hide specifics
- ***Labor-intensive system parts*** *(e.g. UI patterns, web, configuration management, deployment)*
- Dealing with ***feature variability*** and change

A framework can be ***opinionated***, i.e. the framework has certain patterns that you are almost required to use and if you don't, you're gonna have a bad time.

4.5. ARCHITECTURAL VIEWPOINTS

FURPS+ requirements can yield many design issues:

- ***Functionality:*** Might need multiple steps across layers and tiers *(e.g. authentication across multiple layers)*
- ***Reliability:*** Might call for load balancers ore hot/cold standby modes
- ***Performance:*** Hard to judge when only looking at the static structure *(Workload patterns? Single point of failiure? Reaction to error situations?)*
- ***Supportability*** is eased if interactions are easy to understand, test and monitor

Hence, architecture is ***not only about structure***, but also ***behavior*** *(peak/unusual load, error cases)*. Compare to the "Environment" entry in "Quality Attribute Scenarios (QAS)" (Page 7).

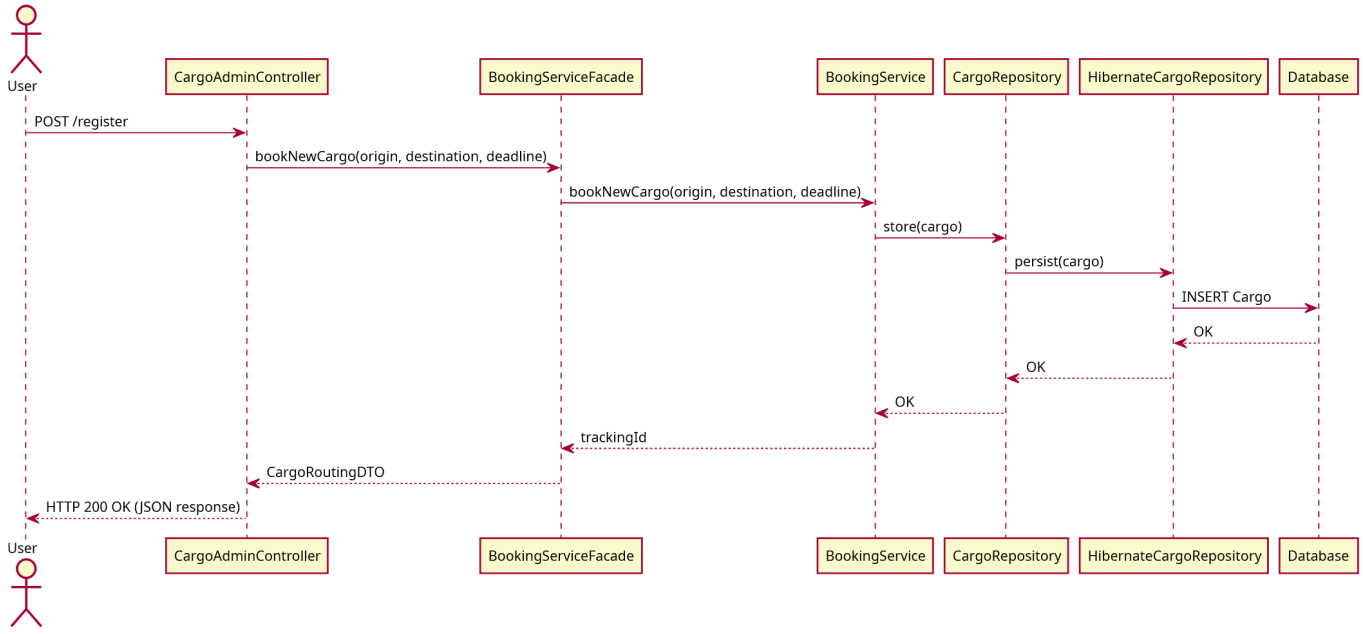
4.5.1. Component Interaction Diagram (CID)

The arc42 documentation template has a ***"Runtime View"***. It describes ***concrete behavior*** of the system's building blocks – i.e. their interactions in the following areas:

- ***Important use cases or features:*** how do building blocks execute them?
- ***Interactions at critical external interfaces:*** how do building blocks cooperate with users and neighbouring systems?
- ***Operation and administration:*** launch, start-up, stop
- ***Error*** and exception scenarios

These points can be notated as an UML diagram, C4 dynamic diagram or a simple list of steps. Often, it can be generated from other artifacts *(i.e. network monitoring)*.

A ***Component Interaction Diagram (CID)*** allows us to reason about miscellaneous qualities, workloads and volume metrics *(i.e. Should access to customer DB be isolated/cached due to frequency? What happens if a network response doesn't arrive on time?)*



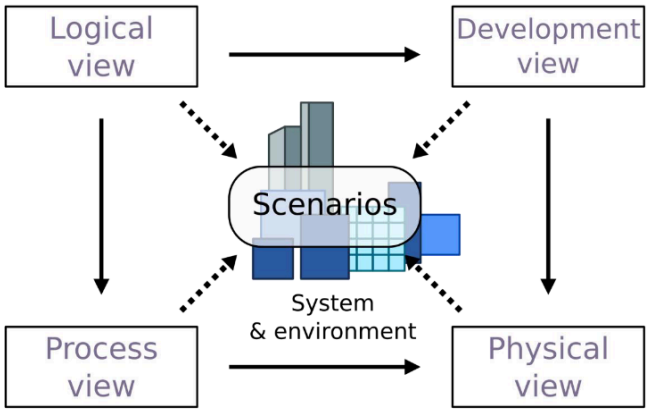
4.5.2. The 4 Views on Software Architecture

- ***Context:*** View the system as a whole and its integration with neighboring systems *(C4 context/system landscape diagrams)*
- ***Building Block View:*** View the static structure of modules and relationships *(C4 component diagram)*
- ***Runtime View:*** How do the individual modules work together? *(C4 dynamic diagram)*
- ***Deployment View:*** In what environment is the system running? *(C4 container/deployment diagram)*

4.5.3. The 4+1 Architectural View Model

4+1 uses similar concepts as the 4 views above, but adapts them to fit with the "+1": Scenarios

- ***Logical View:*** Functionality the system provides to end users *(C4 context diagram, DDD context map)*
- ***Process View:*** Dynamic aspects of the system, system processes and their communication, runtime behavior of the system *(C4 dynamic diagram, CID)*
- ***Development View:*** System from the programmer's perspective, the software management *(C4 components & classes diagram)*
- ***Physical view:*** System from the system engineer's perspective, the topology of the components on the physical layer and their connections with each other
- ***Scenarios:*** Describe interactions between objects and processes *(User stories)*



5. COMPONENT IDENTIFICATION, POEAA, TACTICAL DDD

5.1. COMPONENT IDENTIFICATION

We are currently just operating on the Container/Bounded context level. Everything below is still not organized yet. To carve out responsibilities and create appropriate subcomponents *(modules, projects etc.)*, we use ***Component Identification***.

Recommended methods:

- Story mapping & splitting
- Business process modelling or Collaborative Modelling
- Object-Oriented Analysis and Design (OOAD) patterns
- Derive from use cases/user stories

Still okay methods, but can lead to anti-patterns:

- Use structures of existing structures
- Implement existing industry standard
- Structures suggested by frameworks

5.2. STORY SPLITTING

Used to create, split and prioritize user stories. Closely related to story mapping, see chapter "Story mapping" (Page 27). A good user story should represent a ***vertical slice*** *(a single story should make changes to each layer to deliver an increment of value)* and meet ***INVEST*** properties:

- ***Independent:*** Stories should not overlap and be schedulable and implementable in any order
- ***Negotiable:*** Not everything is set in stone, the details can be changed during development
- ***Valuable:*** Present to the stakeholders why it should be implemented
- ***Estimable:*** Good approximation of the time/budget possible
- ***Small:*** Should fit within a single iteration *(e.g. Sprint)*
- ***Testable:*** In principle, even if the test doesn't exist yet

Best done ***after Domain Storytelling/Event Storming***. They create common understanding about the domain and a structured backlog. With user story mapping/splitting, this backlog can then be partitioned into appropriate stories.

5.2.1. The Five Story-Splitting Patterns

1. ***Split by workflow steps:*** A workflow usually consists of multiple steps that can be split up
   ("publish a news story" becomes "publish a story directly to the website", "publish a news story with editor review", "view a news story on a staging site" and "publish a news story from staging to production")
2. ***Split by operation:*** The word "manage" usually involves multiple actions that can be split up
   ("manage my account" becomes "sign up for an account", "edit my account settings" and "cancel my account")
3. ***Split by business rules:*** A story may have different business rules hidden inside
   ("search a flight with flexible dates" becomes "in N days between X and Y", "on a weekend in December" and "in $\pm$ N days of X and Y")
4. ***Split by variations in data:*** Different variations of a action might need different data
   ("find a route from A to B" is split into "…by car", "…by public transport", "…on foot")
5. ***Split by interface variations:*** Complexity may hide in the interface. Build a simple variant, add fancy later
   ("search for flights in a date range" is split into "…using simple date input" and "…with a fancy calender UI")

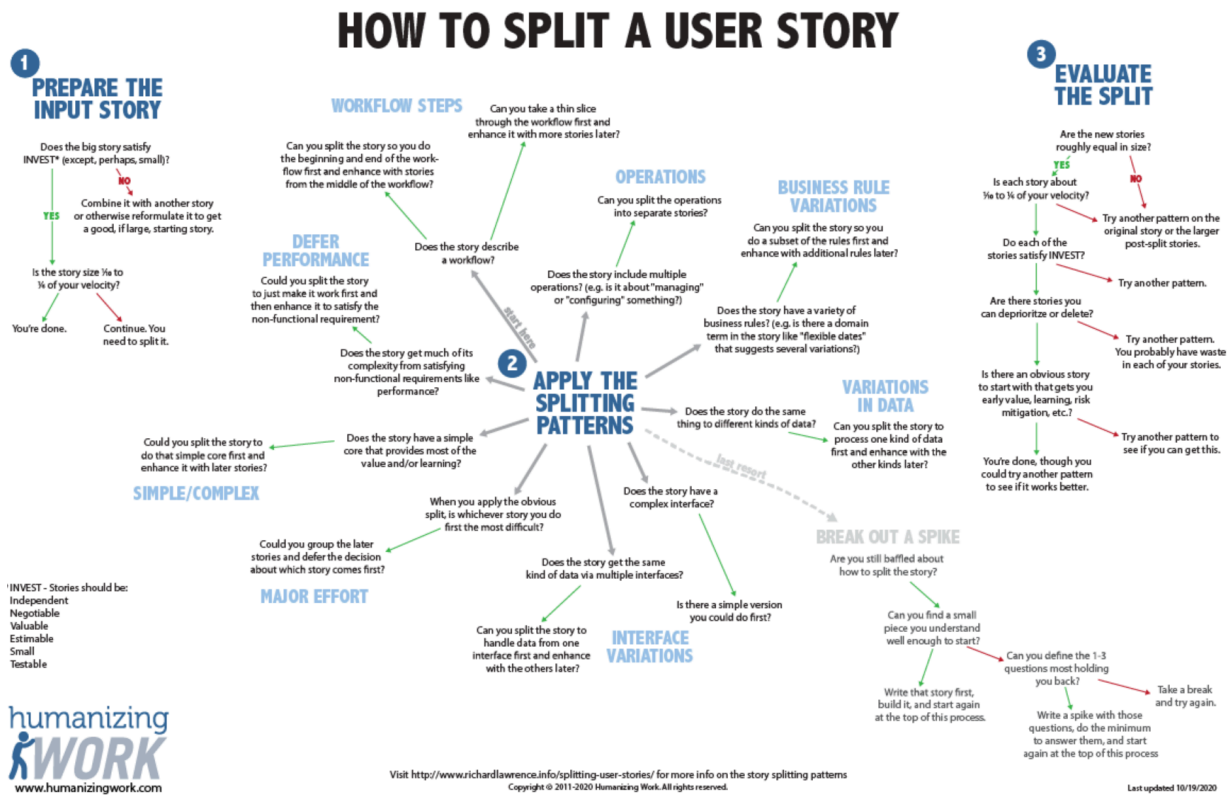
Example of using the Story-Splitting Patterns:
At the start, we have these three Epics. Split them into smaller ones using the Story-Splitting Patterns.

1. **Create Game:** As a developer, I want to create a new game entry so that I can later upload versions and publish it.
2. **Upload Version:** As a developer, I want to upload a new version of my game so that it can be reviewed and published.
3. **Track Status:** As a developer, I want to view the status of my uploaded versions so that I know which ones are under review.

Each split suggests concrete controllers, application services, aggregates/repositories, adapters, and domain events.
(e.g., upload validation → validator service; cross-context publication → event publisher)

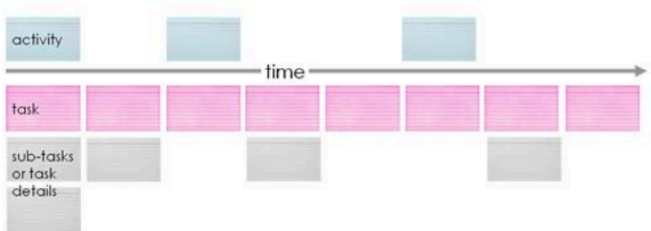
Epic	*Split Story*	*Pattern used*	*Impact on Architecture*
Create Game	Create game with title & description only, saved as draft, add other details later	Workflow step	GameController + GameApplication + GameRepository. Supporting both workflow steps, game aggregate *(state draft)*
Create Game	Add additional data like category, tags, artwork. Depending on category, different details might be needed.	Data variation	Extend Game aggregate, Media Metadata value object, StorageAdapter for artwork
Create Game	Validate title uniqueness	Business rule	Some kind of validation service, maybe domain service?
Upload Version	Upload binary package + version number + changelog	Operation	VersionController → VersionApplication-Service → GameVersionRepository. GameVersion Aggregate
Upload Version	Pre-validate package format/ signature before persisting	Business rule	GamePackageValidator (domain/ service); error events
Upload Version	Trigger Publication/Review request event after successful upload	Interface / Cross-context	EventPublisher → message bus, domain event PublicationRequest
Track Status	Show list of versions with states *(Draft, Uploaded, Publication requested)*	Workflow step	Version read model or query end point, add status field to GameVersion
Track Status	Notify dev on state change *(email/ webhook)*	Interface	NotificationAdapter, subscribe to state-change events
Track Status	Filter by game or state	Data variation	Query parameters, indexes on GameID, Status

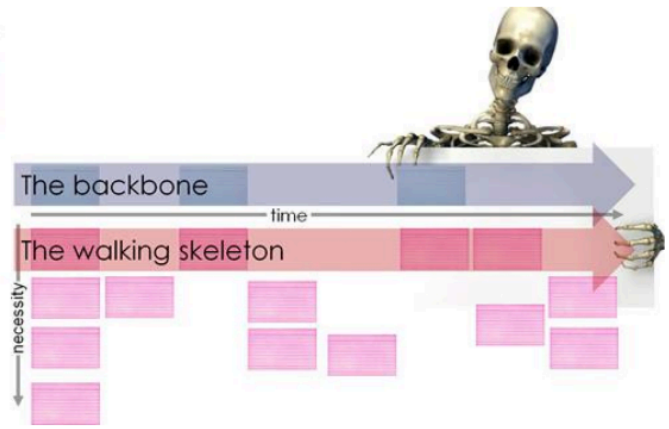
Additionally, there's also this ***monster*** of a graph. We hope we don't need this in the exam. Placed here for reference.



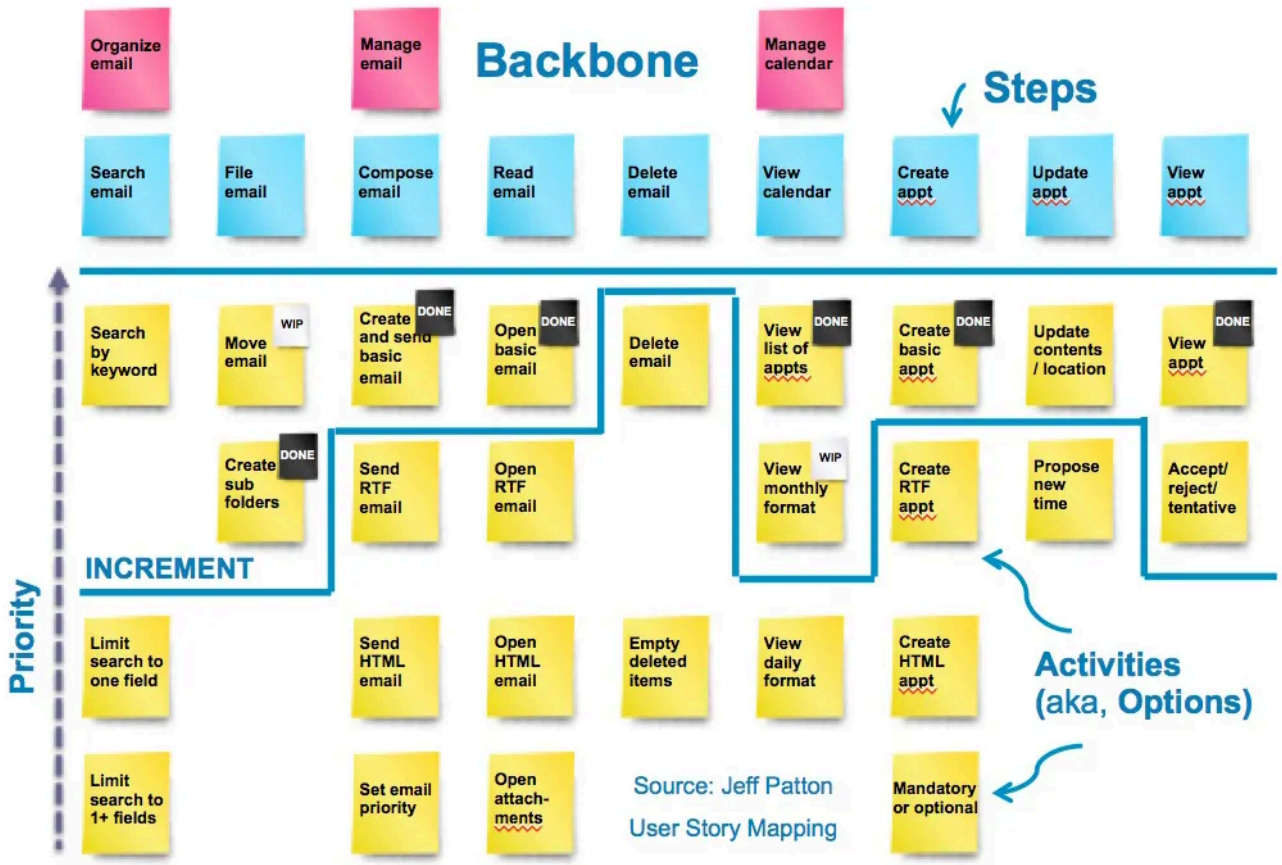
5.3. STORY MAPPING

Closely related with story splitting. Stories should be able to be placed on two axes: ***time*** and ***level of detail***. On the top are the ***activities/epics***, below the ***tasks*** and ***subtasks***. The epics are the ***backbone*** of your project, the ***essential capabilities the system needs to have*** for end to end functionality. The backbone should not be prioritized – it just "is". Only prioritize the stories hanging down from the backbone. The higher they are, the more necessary they are. The task placed the highest therefore represent your MVP, the ***walking skeleton***. Let's call him Gary. Hi Gary!





Example story map:
The ***increment line*** represents what task are planned in this iteration and the already finished tasks. It should move from the top left to the bottom right during a project.



5.4. PATTERNS OF ENTERPRISE APPLICATION ARCHITECTURE (POEAA)

Patterns of Enterprise Application Architecture (PoEAA) is a book from 2003 that contains many important patterns that are still used today:

- ***Domain Model:*** Covered with DDD
- ***Service Layer:*** Covered with Onion/Clean/Hexagonal architectures
- ***Data Mapper:*** Mapping between domain-/data-/API models
- ***Controllers:*** Often seen in Frameworks like Spring
- ***Data Transfer Object (DTO):*** Separate data model for APIs and communication with the outside worlds

While the patterns are still valid, many alternatives have popped up since the book's release *(MVVM, Remote User Interface CSC)*. The business logic layer patterns are rather simplistic and better served with DDD. The Data access layer pattens are mostly implemented within O/R mappers.

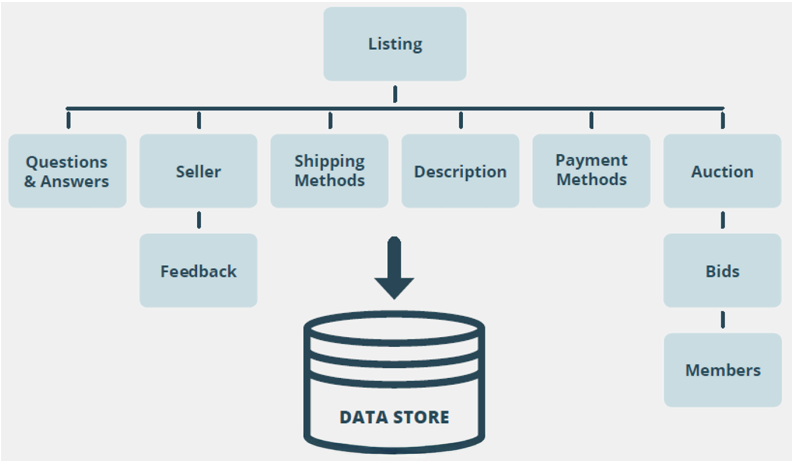
5.5. TACTICAL DOMAIN-DRIVEN DESIGN

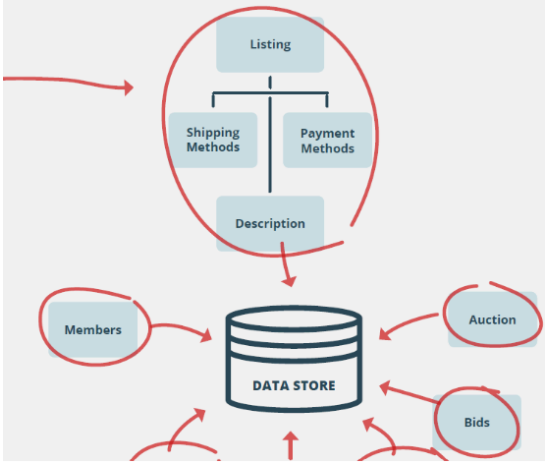
Tactical DDD focuses on the design of the model inside a bounded context – the design of a component, unlike ***Strategic DDD*** which emphasizes the bigger picture, see "Strategic Domain-Driven Design (DDD)" (Page 14).

Term	*Description*
Entity	***Mutable data object***, able to change state. Has an identifier and a lifecycle. *(**Example:** Cargo to be delivered, has fields that change during delivery)*
Value Object	***Immutable data object***. Has no identifier, is defined by its values. All operations must be side-effect free. *(**Example:** ZIP code, doesn't need a separate ID)*
Aggregate	***Collection of entities and value objects.*** Smallest unit with functional consistency *(enforces invariants)*. All objects of an aggregate are persisted as a whole, creating a ***transaction boundary***. Defines a root entity which is the only thing that external entities should reference. *(**Example:** "Customer" aggregate, containing "Address" entity and "Social Security Number" Entity)*
Domain Event	A ***representation*** of ***something that happened*** in the domain. Activity is represented as a series of events. Immutable, as we can't change the past. *(**Example**: "Cargo loaded", when? where? what route?)*
Domain Service	***Domain logic*** that ***crosses*** aggregate boundaries. Contains logic that can't be assigned to a domain object naturally. Is stateless. Two types: ***Domain services*** *(core domain logic)* and ***Application services*** *(don't belong to the domain model, connects infrastructure and domain model)* *(**Example:** Routing service for the cargo.)*
Repository	Handles ***persistence of aggregates***. One repository per aggregate. Doesn't contain business logic. Only the interface belongs to the domain model, the implementation is replaceable and belongs to the infrastructure layer. *(**Example:** CargoRepository that searches for a specific cargo by Tracking ID)*
Factory	The Factory GoF pattern, responsible for ***creation of complex domain objects***/aggregates. Aggregates should be created in one piece ☠ to enforce variants.

5.5.1. Aggregates

Problem: A single object graph may closely relate to the real domain, but may be a bad model. Always treating all information of an object at the same time for different purposes can lead to conflict on unrelated changes.
Example: Asking a question while someone is trying to make a bid.
Solution: Break large objects into smaller ones *(Aggregates)* that are based around invariants/business rules. This ensures operations only get the information they need to perform their function – setting ***transaction boundaries***.





Best practices:

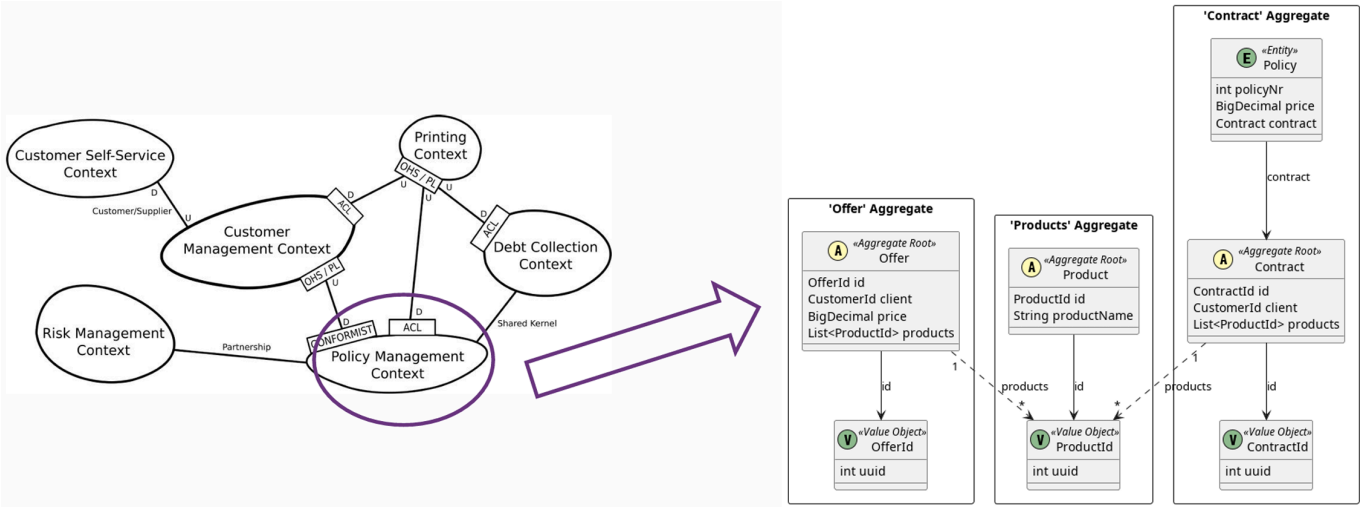
- Use asynchronous communication between aggregates *(e.g. events)*
- The root entity should enforce invariants
- Use the same boundaries for transactions
- Design small aggregates
- Reference other aggregates by identity
- Use eventual consistency *(outside the boundary)*

5.5.2. Invariants

Some examples of invariants as Aggregate/Service cutting criteria are:

- ***Physical containment relationship:*** No “order item” without “order”, “order” with “item x” can’t contain “item Y”, adding “item X” to “order”, lets “item Y” get a 10% discount
- ***Number calculations/value ranges:*** Total sum of X must not exceed value Y, VAT calculation must match product type, sum of all account transfers must always be 0)

5.5.3. Example Domain Model



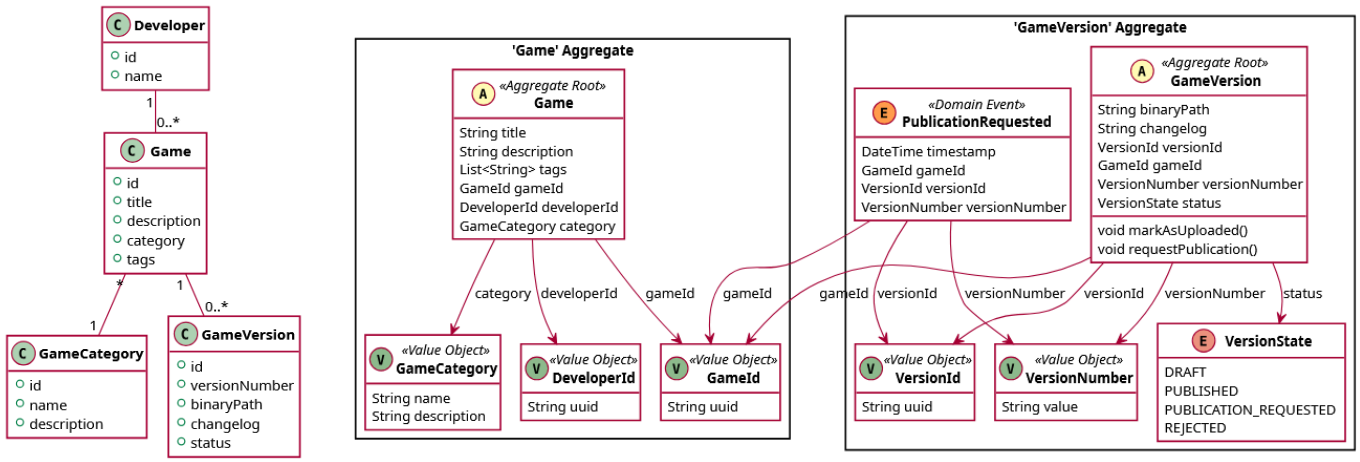
5.5.4. Example: Applying Tactical DDD to Fair Game 3002

On the left is an ***UML diagram*** for creating/uploading games to Fair Game 3002. Refactor it into a tactical DDD model.

Observations:

- The initial model mixes static metadata *(`Game`)* with version-specific data *(`GameVersion`)*. These have different lifecycles and should likely become separate aggregates.
- The Developer entity belongs to another bounded context *(`User Profiles`)*. In this context, it should be referenced only via a `DeveloperId` Value Object.
- There are no clear transactional boundaries *(Aggregates)* or domain events.

Initial Architecture **Solution** *(References between aggregates only use IDs)*



5.6. DOCUMENTING COMPONENTS

Documentation of components should be done with C4, UML or other component diagrams. A popular tool is ***PlantUML***, a language for creating UML diagrams. Because it is text-based, it is more version-control-friendly than purely graphical solutions. It is highly configurable, i.e. to produce C4 diagrams.

These diagrams should usually include:

- ***Controllers:*** Act as entry points for external requests
- ***Application Services:*** Orchestrate use cases *(Depend on domain aggregates and coordinate cross-aggregate operations. Each service typically owns one transaction boundary)*
- ***Domain layer components:*** Aggregates and domain layer services *(Persisted through repositories, though these are usually not shown explicitly)*
- ***Adapters:*** Import, storage, security scanning, event publishing…

5.6.1. Components, Responsibilities, Collaborator (CRC) Cards

Components, Responsibilities, Collaborator (CRC) Cards are a tool to document the purpose of components. It was originally designed for OOP Classes, but it can be repurposed on the architectural level. Each component gets its own card where the ***Responsibilities***, ***Collaborators*** and the ***possible implementation technologies*** are listed.

The text should be keywords and sentence fragments. It can be added to models if written in Markdown etc.

Component: <Name>

Responsibilities:	***Collaborators*** (Interfaces to/from)***:***
– What is this component capable of doing? *(provided services)* – Which data does it deal with? – How does it do its jobs in terms of key system qualities?	– Who invokes this component *(service consumers)*? – Who does this component call to fulfil its responsibilities *(service providers)*? – Any external active/passive connections?

Candidate implementation technologies (and known uses)

- Which technologies, products *(commercial, open source)*, and internal assets can realize the outlined component functionality *(responsibilities)*?

Example:

Component: `VersionApplicationService`

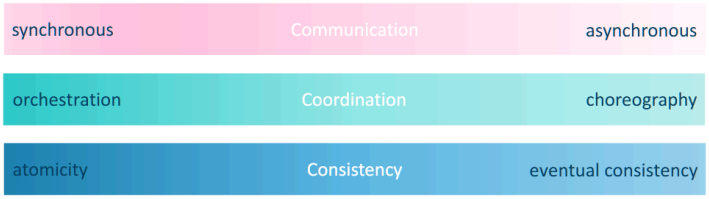
Responsibilities:	***Collaborators*** (Interfaces to/from)***:***
– Coordinate upload workflow for a new game version. – Validate metadata and binary file before persistence. – Interact with the `BinaryStorageAdapter` to store uploaded binaries. – Trigger asynchronous malware scan through `SecurityScanner`. – Maintain transactional consistency within the `GameVersion` aggregate.	– `GameVersion` Aggregate *(domain logic and invariants)* – `BinaryStorageAdapter` *(binary storage)* – `SecurityScanner` *(virus scan trigger)*

Candidate implementation technologies (and known uses)

- Annotated with @Service in a typical Spring Boot application.
- Transactions wrap the aggregate modification and event publication to guarantee atomicity.

6. THE HARD PARTS

Tradeoff Analysis is finding out ***what parts*** are coupled together, ***how*** they are coupled and how to ***assess trade-offs*** by determining the impact of change to interdependent systems.



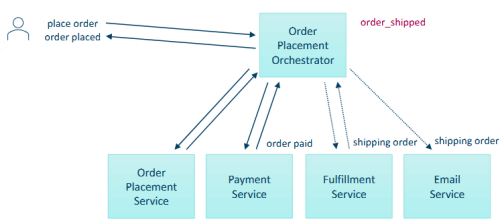
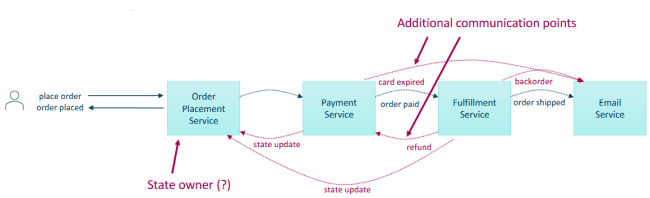
6.1. COMMUNICATION

Communication can be ***synchronous*** or ***asynchronous***. Synchronous leads to ***dynamic entanglement***: The caller waits for a response and blocks everything else. Asynchronous leads to ***loose coupling***, the caller receives response asynchronously and does not block anything.

Synchronous communication	***Asynchronous communication***
+ Easy to model transactional behavior	+ Allows highly decoupled systems
+ Mimics non-distributed method calls	+ Common performance tuning technique
+ Easier to implement	+ High performance and scale
– Performance impact on highly interactive systems	– Complex to build and debug
– Creates dynamic entanglements *(wait for return)*	– Presents difficulties for transactional behaviors
– Creates limitations in distributed architectures	– Error handling needs to cover more cases

6.2. COORDINATION

Should I use ***orchestration*** or ***choreography***? Things to consider: ***Workflow optimization*** *(what fits the use case better)*, ***error handling***, ***state management***.

Orchestration	***Choreography***
An orchestrator ***handles the user input*** and ***the calls*** to the respective services. The services ***don't know anything*** about each other, they just ***communicate with the orchestrator***. The orchestrator ***stores the state*** of the entire workflow.	Each service ***communicates with its neighboring*** service. But in the ***failure case***, a service may have to ***contact another service*** out of line, adding ***additional communication points***. It is also unclear where the ***state*** of the workflow is stored.
+ **Centralized workflow:** Observability and auditing	+ **Responsiveness:** No central bottleneck
+ **Error handling:** Orchestrator handles error states	+ **Fault tolerance:** No single point of failure
+ **Recoverability:** Snapshots and replays possible	+ **Scalability:** Services scale independently
+ **State management:** Single point of truth	+ **Service decoupling:** Only neighbor interactions *(Happy path only)*
– **Responsiveness:** Orchestrator becomes bottleneck	– **Distributed workflow:** Hard to test and monitor
– **Fault tolerance:** Single point of failure	– **State management:** State spread across services
– **Scalability:** Orchestrator hard to scale out	– **Error handling:** Implemented in many places
– **Service Coupling:** Orchestrator knows all services	– **Recoverability:** Each service restores its state
	

6.3. CONSISTENCY

Describes whether the workflow communication requires ***atomicity*** or can utilize ***eventual consistency***.

- ***Atomicity:*** Guarantees that each transaction is treated as a single unit which either succeeds completely or fails completely.
- ***Eventual Consistency:*** Informally guarantees that, if no new updates are made to the given data item, eventually all accesses to that item will return the last updated value *(eventual = schlussendlich, nicht eventuell!)*.

All-or-nothing transaction is one of the ***most difficult problems*** to model in distributed architecture. Avoid cross-service transactions.

6.4. SERVICE GRANULARITY

Choosing the right ***granularity*** *(the size of a service)* is one of the hardest parts in software architecture. It's not ***defined*** by classes or lines of code, but ***by what the service is responsible for*** *(what the service does)*.

6.4.1. Granularity Disintegrators

Provide guidance and justification for when to ***break a service into smaller pieces***.

- ***Service scope and function:*** Is the service doing too many unrelated things?
- ***Code volatility:*** Does only one part of the service change frequently while the rest rarely does?
- ***Scalability and throughput:*** Do parts of the service need to scale differently?
- ***Fault tolerance:*** Are there errors that cause critical functions to fail within the service?
- ***Security:*** Do some parts of the service need higher security levels than others?
- ***Extensibility:*** Is the service always expanding to add new contexts?

6.4.2. Granularity Integrators

Provide guidance and justification for ***putting services back together*** or not breaking them apart in the first place.

- ***Database transactions:*** Is an ACID transaction required between separate services?
  (ACID = atomicity, consistency, isolation, durability)
- ***Workflow and choreography:*** Do services need to talk to one another?
- ***Shared code:*** Do services need to share code among one another
- ***Database relationships:*** Although a service can be broken apart, can the data it uses be broken apart as well?

7. WEB ARCHITECTURE

7.1. SIMPLE HTML

Early web architecture consisted of ***static HTML pages*** served from web servers. ***No dynamic content***, simple request-response only.

Simple Example with Docker and Caddy:

```
# Mount /tmp/web as volume on /usr/share/caddy and redirect connections from port 8080 to 80
docker run -p 8080:80 -v /tmp/web:/usr/share/caddy caddy:latest
```

7.1.1. CGI-BIN

The introduction of CGI *(Common Gateway Interface)* in 1993 led to the first ***dynamic server-side content generation*** with Perl/C scripts to generate HTML, spawning a ***new process per request***.

CGI is an interface specification that enables web servers to ***execute an external program*** to process HTTP user requests. Due to the expensive process launch for each request, it has bad performance.

FastCGI: Variation of CGI with the aim to ***reduce the overhead*** related to interfacing between web server and CGI programs. It has persistent processes and ***reuses*** resources *(database connections, caches)* used in e.g. PHP-FPM *(fastCGI process manager)* for applications like Nextcloud.

7.2. SERVER SIDE RENDERING (SSR)

The server generates HTML/JS/CSS ***dynamically*** and sends the assets in real-time to the browser. PHP, ASP *(Active Server Pages, from Microsoft)* and JSP *(Jakarta/Java Server Pages)* emerged in the mid-1990s and featured better integration than CGI.

7.2.1. Architecture for SSR

SSR uses the ***MVC model***, created in 1979 at Xerox for desktop GUIs:

View / Observer: Renders the ***representation***, ***responds*** to ***changes*** in the model.
Controller: Responds to ***user input***, ***receives*** and ***validates*** input, initiates manipulation of the model.
Model / Subject: Manages ***data*** and ***state*** of the application.

Early web MVC used ***"thin client"*** approach – nearly all Model, View and Controller ***logic ran on the server*** which generated complete HTML pages and sent them to the browser.

Advantages of MVC

- ***Avoids spaghetti code:*** Prevents mixing database/HTML/logic
- ***Separation of Concern:*** Clear responsibilities per component
- ***Model independence:*** Build/test separately from UI
- ***Parallel development:*** Teams work on different layers simultaneously
- ***Easier Testing:*** Straightforward unit tests per component

Well known MVC SSR frameworks

- NeXT WebObjects, 1996 *(first big one, controlled by Apple)*
- Java EE, 1999 *(complex)*
- Spring, 2002 *(Java EE alternative, dominant Java Framework)*
- Spring Boot, 2014 *(Autoconfig for Spring)*
- Ruby on Rails, 2004 *(convention over configuration)*
- Django, 2005 *(Python's Model-Template-View)*
- ASP.NET MVC, 2009 *(Microsoft copying Rails)*
- PHP: Symphony (2005) & Laravel (2011)

7.2.2. Classic SSR Request Flow

```
Request → Router → Controller → Service → Model (DB query) → View Template → HTML Response
```

1. ***User requests URL from Browser:*** Browser sends GET request, TCP connection established, DNS resolves domain
2. ***Server processes request and renders HTML:*** Router receives request and forwards it to the responsible controller. Controller receives it and calls service layer which queries database via repository. Template Engine merges data with HTML template. The complete HTML is generated.
3. ***Complete page is sent to client:*** Server sends full HTML document in response. Separate requests needed for CSS/JS/Images.
4. ***Browser displays final result:*** Parses HTML, builds DOM, applies CSS, calculates layout, renders page and runs JS.

Advantages of classic SSR	*Disadvantages of classic SSR*
– **Fast initial page load:** HTML arrives fully rendered – **Better SEO:** SE can directly parse the complete HTML – **Works without JS enabled:** supports older browsers – **Simple mental model:** clear request/response cycles	– **Full page reload for navigation:** causing screen flashes and losing client-side state – **Server does all rendering work:** Requires more CPU/memory resources per request – **Higher server load with traffic** – **Slower interactions after initial load:** Each click involves a full round-trip to the server

7.3. CLIENT SIDE RENDERING (CSR)

Interactions occur within a single web page, there is no visible "page change" like on regular websites. Client page ***dynamically updates*** as the user interacts with it, providing a smooth, ***app-like experience***. Relies on ***JavaScript*** *(or WebAssembly)* to update the UI.

7.3.1. AJAX (Asynchronous JS and XML)

AJAX allows partial page updates without reload. Start of modern web applications. The server sends minimal HTML shell with JavaScript, Browser executes JS to render UI. Single Page Applications *(SPAs)* enable interaction without page reloads *(e.g. Gmail, Google Maps)*.

7.3.2. Architecture for CSR

CSR uses component-based ***MVVM*** on the frontend and ***MVC*** on the backend. Views now return ***JSON*** instead of rendered HTML, clear separation enables ***independent scaling*** and ***technology choices*** for each layer.
3-Tier: Presentation Tier *(Client-Side)*, Application Tier *(API Server)*, Data Tier *(Server)*.

Frontend – Component-based with MVVM: Model *(data)*, View *(UI)*, ViewModel *(two-way data binding)*. Client-Side routing and state management, API client layer for backend communication.

Backend – MVC Pattern: Model *(data)*, View *(JSON)*, Controller *(handlers)*. RESTful/GraphQL API endpoints serve JSON data. Stateless authentication with JWT tokens.

7.3.3. CSR Request Flow

1. ***User requests URL from Browser:*** Browser sends GET request, TCP connection established, DNS resolves domain
2. ***Server sends minimal response:*** Returns empty HTML shell *(no content)*, includes JS bundle reference
3. ***Browser downloads and executes JS:*** Downloads *(often large)* JS bundle, parses and executes framework code, framework initializes
4. ***JS fetches Data:*** Makes separate API call, API queries DB and returns JSON
5. ***Browser renders UI:*** JS manipulates DOM to build interface, Page becomes visible and interactive *(TTI)*

Advantages of classic CSR	*Disadvantages of classic CSR*
– **Fast interactions after load:** Feels like a desktop app – **Lower server rendering load:** Only servers JSON – **Clear frontend/backend separation**	– **Bundle Size Problem:** Large JS files, slow parsing/execution, mobile struggling – **Slow initial load:** White screen until JS executes – **SEO Problem:** Crawlers see empty HTML, content only after JS execution – **Requires JS to be enabled**

7.3.4. CSR Improvements

– ***Code Splitting:*** Split JS into smaller chunks, load code only when needed *(1 bundle per route)*. Reduces initial bundle size.
– ***Lazy Loading:*** Defer loading non-critical resources, load images on demand. Improves initial page load time.

7.3.5. History of JS Frameworks

– jQuery, 2006 *(simplifies DOM manipulation)*
– Backbone.js, 2010 *(introduces Structure)*
– AngularJS, 2010
– React, 2013
– Vue, 2014
– A million different JS frameworks since then…

7.4. HYBRID APPROACH: HYDRATION

Combines SSR and CSR benefits. Server renders the initial HTML, client hydrates and takes over.

Hydration Process: Server sends ***pre-rendered HTML***, Browser displays content ***immediately***. JavaScript ***loads*** and ***attaches*** event handlers – Application becomes ***interactive***.
Problems: JS re-executes on client – ***duplicated work***, large hydration ***cost delays interactivity*** *(Server Components like in React reduce bundle size sent to client)*. ***Uncanny Valley:*** looks ready but is not yet interactive.

Solution Strategies

– ***Server components (React):*** Components render only on server, no JS sent to client
– ***Island Architecture:*** Static HTML with interactive "islands", only islands ship JS *(Astro)*
– ***Streaming SSR:*** Send HTML in chunks as ready, Browser renders earlier – improves perceived performance *(Qwik)*
– ***Edge Rendering:*** Render closer to user geographically at CDN edge locations – lower latency than origin server

7.5. COMPARING SSR, CSR & HYBRID

Performance Metrics: ***TTFB*** *(Time to first byte)*, ***FCP*** *(First Contentful Paint)*, ***TTI*** *(Time to interactive)*.

Server Side Rendering	*Client Side Rendering*
– ***Excels at FCP and TTI*** *(Visible and interactive immediately)* – Content immediately visible & interactive on load – Page reloads needed for navigation – No *(built-in)* Separation of Concerns – No independent scaling	– Requires full JS execution, bad for FCP and TTI – Must download, parse & execute JS on load – ***No page reloads*** needed for navigation – ***Clean Architecture:*** Frontend / Backend separation *(Static assets on frontend, backend as dedicated API endpoints)* – ***Independent*** scaling and technology choices

When to use what:

Server Side Rendering	*Client Side Rendering*	*Hybrid*
– SEO important – Fast initial page load needed – Content changes frequently – Public-facing marketing sites	– Application behind authentication – Rich interactions more important than initial load – Admin panels, Dashboards	– Need both: SEO + rich interactivity – E-commerce, social platforms – Willing to accept deployment complexity

Takeaways

– ***No single approach fits all:*** Only more/less false. Understand the trade-offs of your requirements.
– ***Your architecture has a huge impact on performance:*** Initial load vs. subsequent interaction, server vs. client load, SEO vs. interactivity
– ***Use common standards and naming:*** Architecture can evolve as requirements become clear. Consistency is more important than perfection *(Easier to understand for others, LLMs and you in 3 months)*.

8. EXAMPLE ARCHITECTURES

8.1. DEPENDENCY INJECTION

Dependency injection (DI) is a programming technique that makes a class ***independent*** of its dependencies. It achieves that by decoupling the usage of an object from its creation: The ***dependency*** is ***not instantiated within the class***, but passed to it as a ***parameter*** or via a ***Dependency Injection container***. This helps to follow SOLID's dependency inversion and single responsibility principles.

Dependency injection follows the ***Inversion of Control principle***. The goal is ***loose coupling*** – the component doesn't care how a dependency implements its features, it just calls the interface.

8.1.1. Different Levels of DI in Spring

Type	*Description*	*Example*

Without DI	Instantiation is hard coded, tight coupling, can't swap implementations or test easily	`public UserController() {`
		`  this.emailService = new EmailService();`
		`}`

Manual Wiring	Controlled creation of dependencies, explicit wiring in Config-Class, full control but verbose	`class AppConfig {`
		`  public ES() { return new EmailService(); }`
		`  public UC() { return new UserController(ES()); }`
		`}`
		`class UserController {`
		`  public UserController(EmailService e) {`
		`  this.emailService = e;`
		`} }`

Type	*Description*	*Example*

Auto-Wiring	`@Service` and `@Autowired/final` *(if the constructor is declared `final`, the `@Autowired` annotation is not needed)*. Spring manages dependencies with constructor injection.	`@Service`
		`class EmailService { /*...*/ }`
		`class UserController {`
		`  private final EmailService e;`
		`  public UserController(EmailService e)`
		`}`

Interface-based	Depend on interface, not concrete class. Swap service by changing annotation *(like in the example)*. Loose coupling with zero controller changes needed.	`interface NotificationService { /* ... */ }`
		`@Service`
		`EmailService implements NotificationService {}`
		`// @Service`
		`SmsService implements NotificationService {}`
		`class UserController {`
		`  private final NotificationService ns;`
		`}`

Profiles	Different profiles for different environments. Set in Spring config: `spring.profiles.active=email`	`@Service`
		`@Profile("email")`
		`class EmailService { /* ... */ }`

Qualifiers	Explicit selection to change between multiple implementation. Fine grained control.	`@Service("email")`
		`class EmailService { /* ... */ }`
		`class UserController {`
		`  public UserController(`
		`  @Qualifier("email") NotificationService ns)`
		`  {} }`

8.1.2. When to use DI

Pros	*Cons*
+ **Loose coupling:** easy to swap implementations	– **Complexity:** Additional Framework Overhead
+ **Testability:** inject mocks for unit testing	– **Learning Curve:** Understanding IoC *(Inversion of Control)* containers takes time
+ **Separation of Concerns:** Components focus on business logic	– **Runtime errors:** Missing dependencies only fail at runtime
+ **Centralized Configuration:** Manage dependencies in one place	– **Debugging:** harder to trace object creation flow
+ **Reusability:** Components can be used in different contexts	– **Overkill** for simple applications

Architecture decisions ***depend*** on the ***Framework***. Each Framework has it's own philosophy and best practices. ***Do not fight against your Framework!*** This creates unnecessary complexity. Use best practices recommended by framework:

– ***Spring Boot:*** use JPA and DI
– ***Go:*** Avoid ORMs and DI frameworks, they are not idiomatic. Instead, use interface-based design.
– ***Middleground:*** Squirrel, Jet, jOOQ

8.2. DATABASE ABSTRACTION

Spring Data JPA *(Jakarta Persistence API)* is a Data Layer framework to simplify hooking up your code with your database *(similar to .NET Entity Framework)*

Spring is easier for DDD implementation. Spring Data JPA implements the Repository pattern directly, see chapter "Tactical Domain-Driven Design" (Page 29).

In ***golang*** DDD is possible, but decoupling Go doesn't always provide benefits because you rarely swap databases.

Reasons to use JPA	*Reasons not to use JPA*
– **Abstraction & Portability:** Switch databases, keep data access code. Standard API *(Hibernate, EclipseLink)*, reduced boilerplate compared to raw JDBC *(Java Database Connectivity)*. – **Domain-Driven Design:** Behavioral domain models with entities as first-class objects, not just DTOs *(Data Transfer Objects)*. Object-oriented query language *(JPQL – Jakarta Persistence Query Language)* instead of table-centric SQL – **Productivity:** Auto-generated basic CRUD operations, management of transactions and connection pooling	– **Problem "entities + N":** Multiplying DB calls *(for a single request, like fetching Users, additional requests for each User to get their information are needed)*. Often the case with Lazy Loading. Requires acquiring knowledge to detect and fix. – **Loss of Control:** Generated SQL may not be optimized. But you can use raw JDBC. – **Complexity & Learning Curve:** Requires understanding of database internals, SQL, and behaviors like flush ordering. Increases dependencies.

8.3. TRANSACTION MANAGEMENT

Ensures that ***multiple database*** operations execute ***atomically***. Prevents ***partial*** updates that corrupt data consistency. Critical for ***maintaining data integrity*** when business logic spans multiple table modifications or when concurrent users access shared data.

Spring Boot	*Golang*
– `@Transactional` annotation magic / rollback rules – **Proxy-based gotchas:** Self-invocation doesn't start transactions, must be `public` methods, only works when called externally.	– Explicit `begin()`, `commit()`, `rollback()` – `defer` statement for cleanup – Context-based transaction passing – Manual transaction boundaries
Convenience: Harder to trace, but rarely issues when implemented correctly. Best practices: `@Transactional` on service layer, not DAO *(Data Access Object)*.	**Explicitness:** Easy to trace, best practice: manual transaction passing.

8.4. DATABASE MIGRATION

With multiple developers and multiple environments, ***versioning*** of database schema becomes important for traceability *(Who added that column? When? Why?)*. Dev, Staging and Production environments must ***stay in sync***. ***Manual*** SQL executions across environments cause ***inconsistencies*** and ***errors***. CI/CD pipelines need ***automated*** database updates alongside code.

There is need to be able to ***undo*** changes when deployments fail. Databases must support ***backward compatibility*** during version transitions.

Only use database migration if you have a team and a prod environment, else skip it. It ***adds complexity*** and ***requires discipline***.

Spring Boot with flyway	*Golang with golang-migrate*
– **Zero config setup** – Spring Boot calls `Flyway.migrate()` **automatically** during application setup. Has built-in locking mechanisms – Undo only in commercial pro version	– **Idempotent:** less error prone / may hide errors – **Workflow:** Create a table, use timestamp prefixes – **Best practices:** Test locally first, store in version control with code, avoid rollbacks by preferring backward-compatible changes, run via CLI in CI/CD not on app startup.

8.5. TESTING

Catch Bugs ***early*** before production, cheaper to fix in development than after deployment. Enable ***confident refactoring:*** Tests act as safety net when changing code structure. ***Document*** expected behavior, tests serve as executable specifications. ***Prevent Regression:*** automated tests catch when new changes break existing functionality. ***Test pyramid:*** many unit tests, fewer integration tests, minimal end-to-end tests. Aim for ***high coverage*** of ***critical business logic***, not 100% coverage everywhere.

- ***Unit tests:*** isolated component testing with mocked dependencies
- ***Integration tests:*** test multiple components together with real database/services

9. MESSAGING

Before 2000, only ***direct point-to-point communication*** was widely used, leading to ***tight coupling*** and ***scaling problems***. In the early 2000s, proprietary messagings formats *(IBM MQ, MSMQ)* were released, but they were expensive and complex. The open source ***RabbitMQ*** was released in 2007, built in the highly distributed Erlang language. ***Apache Kafka*** *(2011)* is optimized for high-throughput streaming. Most cloud hosters also have their own messaging service. RabbitMQ, Kafka and the cloud-native solutions coexist for their specific purposes.

Useful for:

- ***Decoupling:*** Services don't know about each other.
  Trade-off: *Harder to debug distributed flows.*
- ***Asynchronous processing:*** Non-blocking operations.
  Trade-off: *No immediate feedback on success/failure.*
- ***Load leveling:*** Handle traffic spikes.
  Trade-off: *Delayed processing during spikes.*
- ***Reliability:*** Guaranteed delivery, retry mechanisms.
  Trade-off: *Need to handle duplicate messages / idempotency.*
- ***Scalability:*** Horizontal scaling of consumers.
  Trade-off: *More complex deployment and coordination – More costs*
- ***Integration:*** Polyglot systems communication
  Trade-off: *Serialization overhead, schema evolution challenges*

When ***not*** to use messaging: Simple CRUD, low traffic, single monolithic application. Choose based on ***throughput***, ***ordering*** and ***replay needs***. There is no perfect solution. Always start simple, scale when needed.

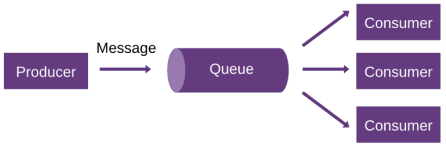
9.1. MESSAGE BROKER PATTERNS

Point-to-Point / Queue

One producer, one consumer. Good for Work Queues.
Example: Payment services sends order to fulfillment service.

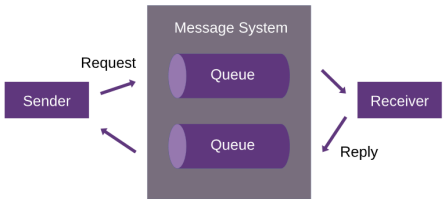
Publish / Subscribe

One producer, many consumers. Good for Event Broadcasting.
Example: Order is created, notify inventory, shipping, analytics, email service.
FairGame: User finished game, need to update multiple systems *(Leaderboard, Achievements, Statistics, Analytics)*



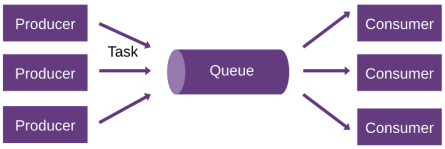
Request / Reply

Synchronous-like behavior over async messaging. Like RPC, but more reliable. Good for internal microservices
Example: Service A asks Service B for data, waits for response.



Work Queue (Competing Consumers)

Multiple workers competing for tasks from same queue. No order guarantees. Tasks must never be deposited twice into the queue!
Example: 5 video encoding workers pulling from encoding queue.



9.2. PUSH VS. PULL MODELS

- **Push:** Broker ***actively delivers*** messages to consumers. Consumer registers callback, broker pushes when message arrives. Consumer needs to be ready at all times. **Examples:** RabbitMQ, AWS SQS.
- **Pull:** Consumer ***actively request*** messages from broker. Consumer controls when and how many messages to fetch *(Batches)*. Higher Latency. Use for e.g. video encoding. **Examples:** Kafka, Amazon Kinesis.

9.3. MESSAGE DELIVERY GUARANTEES

- **At-Most-Once:** Fire and Forget. Message delivered ***0 or 1 times***, never duplicated. Producer sends message, doesn't wait for `ACK`s. ***Use case:*** Monitoring metrics, telemetry, loss acceptable.
- **At-Least-Once:** Messages not lost but may be delivered ***multiple times***. Producer waits for broker acknowledgment, broker persists message before `ACK`, consumer must explicitly acknowledge processing. Majority of production messaging systems. Requires ***idempotent*** consumers. **Examples:** RabbitMQ, SQS default.
- **Exactly-Once:** Expensive Illusion. True exactly-once delivery is ***theoretically impossible*** due to network partitions, crashes, and timing issues. **Reality:** ***At-least-once delivery + idempotent operations*** is effectively "exactly-once". ***Use case:*** Financial transactions, payments.

9.3.1. Idempotency

Same message can arrive multiple times. The same message processed repeatedly must ***produce same result***, this is called idempotency. Track processed message IDs in the database, use natural keys *(e.g. order_id + user_id + timestamp)*. Design your operations to be inherently idempotent.

Code	*Idempotent?*	*Explanation*
`balance = balance - amount;`	✗	Run twice → double deduction
`UPDATE SET balance = 100 WHERE id = X`	✓	Run twice → same result
`INSERT ... ON CONFLICT DO NOTHING`	✓	Upsert with unique constraint

9.3.2. Error Handling & Retry Strategies

- ***Transient Errors:*** Temporary, Self-healing. Retry with exponential backoff + jitter *(random variation between retries)*
- ***Non-transient Errors:*** No retry will fix these.

Dead Letter Queue (DLQ): Special queue for messages that ***can't be processed*** after multiple retries. Messages go to this message heaven if they exceed max retry attempts, if the message TTL expired or when deserialization errors or validation failures occur. Store the message in the DLQ with the error and number of retries. ***Flow:*** Main Queue → fail → Retry Queue *(with delay)* → fail → DLQ

9.4. MESSAGE ORDERING

Most distributed messaging systems do ***not*** guarantee global message order by default. Message order is difficult to keep because of ***concurrency***, ***network delays*** and ***retries***. But most systems don't need strict ordering. ***DLQ queue*** is inherently ***out of order***.

9.4.1. Ordering Guarantees

- ***No Ordering:*** Messages may arrive in any order. Default in most systems.
- ***Per-key/Partition Ordering:*** Messages that share a key are delivered in the same order as they were produced.
- ***Single Consumer Ordering:*** Single consumer reading a stream processes messages one-by-one in read order.
- ***Sequence Numbers:*** Each message carries a number so receivers can detect gaps and restore the order.

9.4.2. Common Anti-Patterns

- ***Don't ignore failed messages:*** Use DLQ to prevent them from disappearing
- ***Infinite retries:*** Set a max retry limit to avoid blocking the queue forever
- ***Synchronous processing:*** Defeats the purpose of async messaging
- ***Large messages:*** Slow serialization, memory pressure, network congestion. Use references/pointer instead
- ***No monitoring:*** Track queue depth, processing time, error rates
- ***Assuming order:*** Design for unordered unless explicitly guaranteed
- ***No idempotency:*** At-least-once requires idempotent consumers
- ***No message schema/versioning:*** Allows for easier handling of schema changes

9.5. COMPARISON MESSAGE BROKERS

Feature	*RabbitMQ*	*Kafka*	*ZeroMQ*	*PostgreSQL*
Architecture	Traditional broker *(Erlang)*	Distributed log *(Java/Scala)*	Brokerless library	Database-based queue
Throughput	Medium	High	High	Low
Latency	Low	Medium	Lowest	Medium
Durability	Memory + Disk	Disk *(configurable retention)*	None	Disk *(ACID – Atomicity, Consistency, Isolation, and Durability)*
Message Replay	No	Yes *(consumer controls offset)*	No	Possible
Routing	Advanced *(exchanges, topics)*	Simple *(partitions by key)*	Manual *(App-level)*	None
Ordering	Per-queue	Per-partition *(strong)*	No	No
Ops Complexity	Medium	High	Low	Low *(Already using PG)*
Use Case	Task queues, microservices, RPC	Event streaming, analytics, logs	Low-latency, embedded	Simple queues, low volume

9.5.1. Key Differentiators

Rabbit MQ	*Kafka*	*ZeroMQ*	*PostgreSQL*
– Push Model: broker delivers to consumers – Written in Erlang – AMQP protocol, flexible exchanges	– Pull model: consumers fetch messages – Append-only log architecture – Messages persist, enable replay – Built for LinkedIn's scale	– No broker = no single point of failure – "Sockets on steroids": just a library – Microsecond latency, but no guarantees	– `SKIP LOCKED` prevents duplicate consumption – Visibility timeout + retry logic – Leverages existing ACID guarantees – Disk I/O bottleneck

10. MICROSERVICES & REST

Before REST, there was ***monolithic architecture***. This led to tightly coupled architecture. If you wanted to deploy / scape one thing, you hade to deploy / scale everything. Very ***inefficient***.

RPC, CORBA *(Common Object Request Broker Architecture)*, ***SOAP/WS-**** *(Simple Object Access Protocol Web Services)* tried to fix this, but they pretended the network didn't exist; they ran remote functions like local ones. ***Difficult to debug***, distributed computing fallacies applied. ***CORBA:*** high complexity, inconsistent implementations, poor interoperability. Solutions got ***more complex*** than necessary.

Context for REST: The Web thrived with simple HTTP while enterprise systems had high complexity.

10.1. REST FOUNDATIONS

Rest is an ***architectural style***, not a protocol or standard.

10.1.1. 6 constraints of REST

- **Client-Server:** ***Separation of concerns***. Client handles presentation, server handles business logic. ***Independent evolution*** possible.
- **Stateless:** Each request contains all needed info. Server ***doesn't store client state***, each request must bring everything with it. ***Easy scaling***, no session sharing.
- **Cacheable:** Responses must define themselves as (non-)cacheable. ***Reduces network traffic***, ***optimizes performance***.
- **Layered system:** Client can't tell if connected directly to end server. Proxies may be located between the client and server, client doesn't need to be aware of this. This simplifies ***scaling*** and ***security***.
- **Uniform interface:** Resources identified in request URL, ***self-descriptive*** messages.
- **Code-on-demand:** Servers can optionally ***extend client functionality*** by sending executable code to it.

10.1.2. HATEOAS

Hypermedia As The Engine Of Application State is a ***self-describing API*** that directly outputs links to additional related resources *(e.g. a User object contains links to its orders, friends etc.)*. Not widely used, but interesting concept. Complex.

10.2. COMMON MISCONCEPTIONS

- ***JSON ≠ REST:*** JSON is just one representation, REST is format-agnostic.
- ***HTTP ≠ REST:*** HTTP is an implementation of REST, but REST is an architectural style independent of protocol.
- ***HTTP + JSON ≠ RESTful:*** You need to adhere to the constrains for your software to be RESTful *(i.e. statelessness)*.

10.3. RESTFUL HTTP

Most HTTP APIs are only REST-like; they don't follow all REST principles.

10.3.1. HTTP methods

An HTTP method is ***safe*** if it doesn't alter the state of the server and ***idempotent*** if the intended effect on the server of making a single request is the same as the effect of making several identical requests. A ***cacheable response*** is an HTTP response that can be cached, that is stored to be retrieved and used later, saving a new request to the server.

Method	*Safe*	*Idempotent*	*Cacheable*	*Description*
`GET`	✓	✓	✓	Requests a resource
`HEAD`	✓	✓	✓	Same as `GET`, but without response body
`OPTIONS`	✓	✓	×	Describes communication options for target resource
`TRACE`	✓	✓	×	Performs message loop-back test
`PUT`	×	✓	×	Replaces target with the request content
`DELETE`	×	✓	×	Deletes the specified resource
`POST`	×	×	~	Submits an entity to the specified resource *(Side effects on server)*
`PATCH`	×	×	~	Applies partial modifications to the resource
`CONNECT`	×	×	×	Establishes a tunnel to the server

10.3.2. Status Codes

HTTP response status codes ***indicate*** whether a specific HTTP request has been ***successfully*** completed.

Classes:

- ***100 - 100:*** Informational responses
- ***200 - 299:*** Successful responses
- ***300 - 399:*** Redirection messages
- ***400 - 499:*** Client error responses
- ***500 - 599:*** Server error responses

Examples:

- ***200*** OK
- ***201*** Created *(with Location header)*
- ***301*** Moved Permanently
- ***400*** Bad Request
- ***401*** Unauthenticated
- ***403*** Forbidden
- ***404*** Not Found
- ***418*** I'm a teapot
- ***500*** Internal Error
- ***503*** Service Unavailable

10.3.3. HTTP headers

Let the client and the server pass ***additional information*** with a message in a request or response.

Header(s)	*Purpose*	*Example(s)*
Content-Type / Accept	Used for content negotiation. ***Content-Type*** indicates the media type of the resource, ***Accept*** informs the server about the types of data that can be sent back.	`Content-Type: text/html; charset=utf-8` `Accept: <media-type>/<MIME_subtype>`
ETag / If-None-Match	ETag is a ***unique string*** identifying the version of the resource. Conditional requests using If-Match and If-None-Match use this to change the behavior of the request. ***Used to update caches.***	`ETag: W/"<etag_value>"` `If-None-Match: "<etag_value>"`
Cache-Control	***Directives*** for ***caching mechanisms*** in both requests and responses.	`Cache-Control: max-age=180, no-cache, ...`
Retry-After	Indicates ***how long*** the user agent should ***wait*** before making a follow-up request. Usually sent with 429/503 status code.	`Retry-After: <http-date>` `Retry-After: <delay-seconds>`

X-RateLimit-*	Headers for rate limiting.	`X-RateLimit-Limit: ...`
		`X-RateLimit-Remaining: ...`
		`X-RateLimit-Reset: ...`

10.3.4. API Versioning

There are several common ways to version an API:

- ***URL-based versioning:*** E.g. `/v1/ ...` Pragmatic, widely used because it's simple, explicit, easy to route and test. But not fully REST because the URL should identify the resource, not the API version.
- ***Header-based versioning:*** E.g. `X-API-Version: 1`. Aligns better with REST principles because the resource URL stays stable. But less visible and more difficult to test.
- ***Content negotiation:*** E.g. `Accept: application/api.v1+json`. The most RESTful approach, but also the most complex. Not widely used.

10.3.5. API Documentation

Good documentation is essential so users can understand and adopt the API correctly. ***OpenAPI*** is the standard for describing REST APIs. Error cases should also be documented! There are two approaches:

- ***Design-first approach:*** Write the specification before writing the code. Results in a single source of truth, but code and specification can drift apart.
- ***Code-first approach:*** Write code with annotations, generate the specification automatically. Documentation always in sync with code. OpenAPI/Swagger generates documentation for your API consumers.

10.3.6. Authentication / Authorization

Authentication is the identification of the user while ***authorization*** determines which resources an authenticated user is permitted to access.

- **Basic Auth:** Transmits the credentials in the `Authorization` HTTP header with `username:password` in base64 encoding. Does ***not provide encryption for credentials***, only formatting for HTTP Header. Should ***only be used with HTTPS***, otherwise login credentials are transmitted in the open. Used for internal tools, admin interfaces, content that is not public.
- **JWT (JSON Web Token):** Stateless, self-contained, signed token with claims. ***Issued at login*** and ***sent with every request***. Scales well, but is hard to revoke: ***Token is valid until it expires***, which is problematic if it gets stolen.
- **OAuth2:** Grants access ***without*** giving out ***passwords***. Uses short-lived access tokens and optionally long-lived refresh tokens. Standard for modern APIs. Can use JWT or Basic Auth. Simpler than OAuth1.

10.3.7. Rate Limiting

Limits the number of requests per time window to prevent API abuse. If the limit is exceeded, the server returns ***HTTP 429 (Too Many Requests)***. Common response headers are:

Non standard header (widely used, but differing implementations)	***Standard headers*** (currently being standardized)
– `X-RateLimit-Limit:` Max requests – `X-RateLimit-Remaining:` Requests left – `X-RateLimit-Reset:` When the limit resets – `Retry-After:` When the client should retry	– `RateLimit-Limit:` Requests quota in the time window. – `RateLimit-Remaining:` Remaining requests quota in the current window. – `RateLimit-Reset:` Time remaining in seconds in the current window – `RateLimit-Policy:` Contains a quota policy, defined by the server, that clients can use to control their own amount of requests to the server. Contains name of the policy, the limit and the window size.

Browsers don't automatically obey Retry-After, but API clients should.

10.3.8. Best Practices for API Design

Use ***nouns***, not verbs *(`/users not /getUsers`)*, ***Plural*** for collections, ***Hierarchies*** for relationships *(`/users/42/orders`)*, ***Meaningful error messages*** in body, ***test*** with ***real*** HTTP clients.

10.4. REST / MICROSERVICES

REST's simplicity enabled Microservices. Now we can use HTTP + JSON instead of CORBA/SOAP/WDSL which are very complex. No special libraries, IDL *(interface definition language)* compilers, or ORBs *(Object Request Broker)* needed. ***Every*** language/framework can make HTTP requests.

REST is the ***communication enabler***. Made it practical to break systems into many small services. Service-to-service ***communication*** became ***simple*** because of REST.

REST and Microservices have a ***symbiotic relationship***. REST provided the ***communication protocol*** for Microservices. But beware: ***Easy communication does not automatically mean easy architecture.*** Communication is solved, but everything else gets more complex.

10.4.1. Microservices Challenges

- ***Distributed system problems:*** Network latency, partial failures, CAP theorem *(Distributed data cannot simultaneously be consistent, available and partition tolerant)*, everything covered in our DSy Zusammenfassung 😉
- ***Data Consistency:*** Accept eventual consistency *(Data will be consistent eventually)*, Saga pattern with compensating transactions *(Sequence of local transactions, on failure compensating transactions get run to undo changes)*, Idempotency through transaction ID for retries, 2PC *(two-phase commit)* is slow and blocks resources.
- ***Service discovery, load balancing:*** More complex, services need to be found *(via Traefik/caddy/nginx/HAproxy)*
- ***Observability:*** Logging, tracing and metrics needs to be collected from all microservices *(three pillars)*
- ***Deployment complexity:*** Container orchestration, Blue-green deployments *(Two environments, current prod and current prod with new release. Test, then switch)*, canary releases *(Release new version to a small group of users first)*.
- ***Testing complexity:*** Integration testing across services are complex
- ***API Gateway Pattern:*** Single entry point *(Client only communicates with gateway)*, cross-cutting concerns are implemented on gateway *(auth, logging, rate limiting need to be implemented on every client)*. Gateway is single point of failure.
- ***Circuit Breaker Pattern:*** Preventing cascading failures by stopping calls to an unhealthy dependency after repeated errors, returning a fallback and periodically retrying to see if the service has recovered. Often already included.
- ***Operational overhead:*** More things to monitor, deploy, secure. Increased infrastructure costs, need for DevOps/SRE expertise.

10.5. MODERN LANDSCAPE & ALTERNATIVES

Modern system architectures are diverse: ***GraphQL*** fits data-driven apps, ***gRPC*** is used for high-performance service-to-service communication, ***event-driven designs*** enable loose coupling, and ***monolith-first*** / Modular Monolith often works best for small projects.

There is no one-size-fits-all: The right choice depends on the use case, team size, scaling needs, and acceptable complexity. The key is to understand the trade-offs, choose deliberately, and be ready to evolve the architecture as requirements change.

Alternatives to REST	*Alternatives to microservices*
– ***GraphQL:*** Solves the over/under-fetching problem *(REST returns everything associated to an endpoint or needs multiple requests for getting the content)*. But N+1 Problem, every query can return different content. – ***gRPC:*** When REST isn't enough. Has better performance, is binary. Good for service-to-service communication. – ***Event-driven architectures*** via message queues. Decoupled, but more complex. – ***WebSockets / Server-Sent Events (SSE):*** Real-time communication, bidirectional. – ***tRPC:*** End-to-end type safety for TypeScript. Needs monorepository or sharing of types via npm packages, TypeScript only.	– ***Monolith-first approach:*** Start simple, extract services later when needed. – ***Modular monolith:*** Well-defined boundaries, can extract to services later if needed. – ***Serverless / FaaS:*** Functions as a Services, function runs on server on event, only pay when function is run *(AWS Lambda, Azure Functions)*

11. PROTOCOLS

11.1. CUSTOM PROTOCOLS

Designing a custom protocol needs ***more time*** to develop and test, but it can be ***more efficient*** *(space/performance)*. There are existing ***protocol generators*** like Thrift, Avro, ProtoBuf. They work with an ***IDL*** *(Interface description language)* to generate code. They are ***standardized***, but have ***more overhead***.

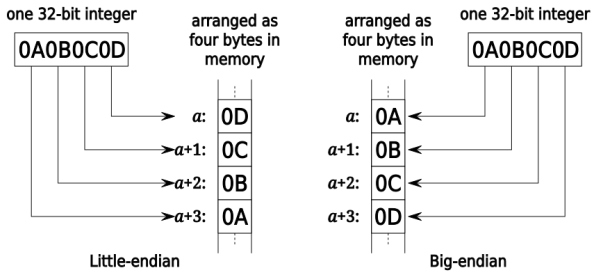
11.2. SERIALIZATION FORMATS

With ***custom encoding/decoding*** you can control ***every aspect***. But this needs time to develop, test and maintain.

Little-endian and big-endian describe the sequential order in which bytes are converted into numbers.

– ***Little-endian:*** Most CPU Arches e.g. x86, ARM, RISC-V.
– ***Big-Endian:*** Networking, e.g. TCP headers

1:1 copy formats like Flatbuffers or Cap'n Proto use the same endianness on the wire and network.



11.3. PROTOCOL EXAMPLES

	ASN1	*Avro*	*Protocol Buffers* (ProtoBuf)
Description	Standard IDL for defining data structures that can be serialized/deserialized.	Data serialization system, RPC framework	Data serialization system from Google *(designed to be smaller/faster than XML)*.
Used in	e.g. X.509 *(TLS certificates)*	e.g. Hadoop *(big-data framework)*	Nearly all inter-machine communication at Google
Data format	Generic binary protocol	Message defined in JSON or IDL – no code generation	Integers to identify fields, Contain only numbers, not field names
Example payload size *(compared to 48 bytes in XML)*	21 bytes	16 bytes *(assuming both have the same IDL)*	18 bytes

11.4. JSON EXAMPLE

JSON + REST/HTTP: Human readable text to transmit data. Often used for web apps. ***More bytes*** than gRPC / Trift because field names are transmitted as text, but there is no schema required at wire level as it has a self-describing format.

+ Universal support, easy debugging
+ No code generation required
− Larger size, slower parsing
− No compile-time type safety

11.5. RPC EXAMPLES

	gRPC	*Thrift*
Description	RPC framework using HTTP/2 transport and Protocol Buffers.	RPC framework from Facebook using an IDL and a binary protocol.
Features	Authentication, bidirectional streaming + flow control, blocking or nonblocking bindings, cancellation + timeouts, many languages.	Cross-platform support across many languages
Example size	171 / 124 bytes *(Wireshark measurement)*	49 bytes transferred *(Thrift encodes which function to call, larger size)*

12. CLOUD NATIVE ARCHITECTURE

Generally a ***trade-off*** between ***flexibility*** and ***complexity***.

Traditional Architecture	***Cloud native Architecture***
– Monolithic applications	– Distributed microservices
– Single database instance	– Multiple specialized data stores
– Vertical scaling *(bigger servers)*	– Horizontal scaling *(More instances)*
– Static infrastructure	– Automated CI/CD pipelines
	– Dynamic infrastructure *(containers, orchestration)*
	– Rapid releases *(hours/days)*

12.1. CLOUD NATIVE CORE PRINCIPLES

The ***12-Factor app methodology*** from 2011 is still relatively correct. But adapt it to your use!

1. ***Codebase:*** One codebase per service in version control
2. ***Dependencies:*** Explicitly declare and isolate dependencies
3. ***Config:*** Stored in environment variables
4. ***Backing services:*** Treat backing services as attached resources
5. ***Separation:*** Strictly separate build and run stages
6. ***Stateless processes:*** Execute the app as one or more stateless processes *(No data in the container itself)*
7. ***Port binding:*** Export services with port binding
8. ***Scaling:*** Scale out with the process model
9. ***Robustness:*** Maximize robustness with fast startup and graceful shutdown
10. ***Parity:*** Keep development, staging, and production as similar as possible
11. ***Logs:*** Treat logs as event streams
12. ***Admin tasks:*** Run admin and management tasks as one-off processes

– ***Containerization:*** Docker/OCI containers is standard in many cases. Immutable artifacts, consistent environments.
– ***Orchestration:*** Kubernetes for container management, auto-scaling based on load, self-healing *(automatic restarts)*, service discovery

12.2. SELF HOSTING – TRADITIONAL APROACH

On-prem infrastructure: ***Physical*** servers in data centers, ***manual*** hardware provisioning. ***Fixed capacity*** – you need to ***over-provision*** for peak load. Direct control over hardware.
Operational reality: 24/7 hardware ***maintenance***, ***manual*** OS patching and updates, backup/recovery ***complexity***. Often ***underutilized*** resources *(typical 10-30% utilization)*.
Costs: ***High CapEx*** *(Upfront investment in long-term assets)* , ***Predictable OpEx*** *(Ongoing running costs to operate and maintain systems)*, ***Staff overhead*** *(sysadmins, network engineers)*

12.3. SELF HOSTING – CLOUD NATIVE APPROACH

Kubernetes on bare metal: Container orchestration on your hardware, ***software-defined infrastructure***. This still requires ***hardware*** management. ***Hybrid model:*** self-hosted control plane *(Cloud orchestration software)*, cloud workers.
Why self-host cloud native: ***Data sovereignty/compliance*** requirements. ***Cost savings*** at scale *(>100 servers)*. Existing infrastructure investment, specific hardware needs *(GPUs, specialized storage)*
Reality Check: Kubernetes ***complexity*** is real, requires specialized expertise. Updates and ***security patching*** is still ***manual***. ***Not*** always ***cheaper*** than cloud *(Consider total cost)*.

12.4. CLOUD NATIVE DATA MANAGEMENT PATTERNS

– ***Database per Service:*** Each ***microservice*** owns its data. ***Independent*** schema evolution, technology fit for purpose *(e.g. SQL, NoSQL, graph)*. But: How to query across services?
– ***Event Sourcing:*** Store events, not current state. Append-only log of changes. ***Rebuild*** state by ***replaying*** events. ***Use case:*** Audit trails, temporal queries.
– ***CQRS:*** Command Query Responsibility Segregation. ***Separate*** write model from read model and optimize each independently. ***Example:*** Write to normalized DB, read from denormalized cache.
– ***Eventual Consistency***: ***Accept*** temporary ***inconsistency***. Systems converge to consistent state over time. ***Trade-off:*** Complexity vs. availability.

12.5. TRADE-OFFS AND CONSIDERATIONS

Traditional	*Cloud Native*
+ Lower complexity	+ Better scaling and flexibility
− Less flexibility	− Adds operational complexity − Operational overhead: monitoring, tracing, logging − More services = more failure modes − Cost: Cloud bills can be high, idle containers still cost money, self-hosting requires expertise.
Use for: – ***Small applications*** with predictable load – ***Small*** development team – ***Simple*** data models – ***Low*** deployment frequency acceptable	**Use for:** – Need to scale specific services ***independently*** – ***Unpredictable*** load patterns – ***Large*** distributed teams – ***Frequent*** deployments *(multiple times per day)* **Rule of thumb: Cloud native pays off at scale.**

13. API DESIGN

13.1. ERROR REPORT PATTERN

Use a ***standardized error format*** *(e.g. JSON or XML)* across ***all endpoints*** so clients can implement ***one consistent error-handling*** approach. A ***universal schema*** reduces parsing effort and enables ***automatic*** monitoring and alerting.

Machine-readable vs. human-readable

Provide both, but keep them separate.

– ***Error codes for client logic:*** Differentiated retry logic, specific UI reactions. Should be consistent, stable and documented.
– ***Messages for Users:*** Should be clear and understandable, safe to show to end users.

Error codes vs. HTTP status codes

Use ***HTTP status codes*** for transport/protocol-level *(e.g. 400, 404, 500)* and ***App codes*** for business logic *(e.g. insufficient funds, user not verified)*. This separation improves error diagnosis.

The ***RFC 7807 Problem Details format*** is a ***standard structure*** for error responses. It defines ***common fields*** like `type`, `title`, `status`, `detail` and `instance`. It is a ***machine-readable*** structure for ***automated error handling***, but sadly not very often used.

Field-level validation errors

Validation errors should be ***mapped*** to the ***specific input fields*** that failed, so the API clearly states ***which field caused which error***. Used with serialization / deserialization.

– ***Backend validation:*** Mandatory as the only reliable security layer
– ***Frontend validation:*** Is optional, used for UX. Should always be consistent with the backend validation.

Actionable error messages for clients

Errors should explain ***what went wrong*** and ***how to fix it***. Generic messages like “500 error” are useless, messages that ***explain*** the error *(e.g. “Gateway timeout – retry after 30 seconds”)* or machine-actionable responses *(`Retry-After` header)* are far more ***useful***.

Error correlation IDs for tracing

Always include a ***unique ID*** per request for log correlation. This is ***essential for debugging*** distributed systems and enables end-to-end ***request tracking*** when users report an issue.

13.2. ERROR LOGGING

Error logging is the foundation for operations and debugging. ***Always log***, and include enough context to reconstruct what happened.

Handle and ***test*** error logging for requests, database, business logic, network, disk, …
For each scenario: add specific error code, clear messages, sufficient context. Systematically test the error handling *(not only the happy path)*.

Log levels

- ***`INFO`:*** normal operational events *(e.g. successful requests)*
- ***`WARN`:*** unusual conditions that could become errors *(e.g. slow DB, degraded state)*
- ***`ERROR`:*** failed operations *(e.g. failed requests, external API failures)*
- ***`FATAL`:*** critical failures threatening system availability *(e.g. lost DB connection, loss of critical resources)*

Set the log level per environment. The ***DEV environment*** can run in the level ***DEBUG***, the ***LOCAL environment*** in the level ***TRACE*** *(max detail)*, while ***TEST/STAGE and PROD*** should run with the level ***INFO***. Try to ***balance*** signal-to-noise-ratio, performance and diagnosability.

Good logging practices

- ***Use meaningful messages:*** Write what exactly went wrong and how to fix it.
- ***Never log sensitive data outside local/dev:*** Passwords, API keys, personal data, pictures of your mum. Assume that logs can be read by every employee.
- ***Include all relevant data:*** correlation ID *(in distributed systems)*, timestamp, HTTP status code / error code, request path / method, client IP / user ID *(if available)*, detailed error message or stack trace *(but don't return stack trace to user)*.

Structured logging: ***Prefer structured logs in JSON for larger Projects.*** There are ***tools*** for this which enable ***search***, ***analysis*** and ***alerting***, but this is ***overkill*** for most small projects.

13.3. RATE LIMITING

Rate limiting ***protects*** services from ***overload*** and ***abuse***. A ***single faulty client*** mustn't degrade availability for ***all***.

Algorithms

- ***Token Bucket / Leaky Bucket:*** Steady outflow, allows traffic bursts as long as the average rate stays within limits.
- ***Sliding Window:*** More accurate rate enforcement, continuous counting instead of fixed intervals.

Return ***`HTTP 429 Too Many Requests`*** when the limit is exceeded so clients can react. This is required for compliance with HTTP semantics.

Standard Headers: `X-RateLimit-Limit`, `X-RateLimit-Remaining` *(How many requests are left)*, `X-RateLimit-Resest` *(How long until the limit resets, mainly used with window-based limits)*. This enables ***intelligent retry scheduling***.

Graceful degradation

Fallback responses or reduced functionality, ***maintain partial service during overload***. Prioritize ***critical*** endpoints *(e.g. login)* ***before low-priority*** read-only endpoints *(e.g. analytics dashboard)*. Return reduced/summary data if needed.

Monitoring and Alerting

Track sustained rate limit breaches to ***detect abuse*** or ***buggy clients*** early. Trigger ***incident response workflows*** if necessary *(alerts, automatic blacklisting, contact customer regarding legitimate misuse)*.

Rate limit can be implemented in different layers:

- ***Application-level:*** Middleware or directly in the service/backend. More flexible for business rules, but requires code changes.
- ***Infrastructure-level:*** Reverse Proxy, API Gateway Tools. Centralized interface before backend. No code changes, but less flexible for business logic.

Best practices is a combination: Rough limitations on the gateway, fine-grained rules in the service.

```
{
  auto_https off
}
:8080 {
  rate_limit {
    zone api_zone {
      key {remote_host}
      events 100
      window 1m //window rate limiter
    }
  }
  file_server
  root * /srv
}
```

13.4. PAGINATION

Pagination is essential for large APIs. Without it, clients might need to download ***millions*** of records at once, causing ***poor performance*** and ***bad UX*** on the client side and high unnecessary resource use on the server side.

Offset vs. Cursor

Both solve paging, but differ in performance, reliability and complexity. In most cases, cursor pagination is better.

- **Offset pagination:** Client sends `offset + limit` *(e.g. fetch rows 101-110)*. This leads to ***performance issues***, because the DB needs to load, scan and skip the first 100 entries which has a performance of $O(n)$. There are also ***consistency issues*** with dynamic data because inserts/deletes can cause duplicates or missing records between pages.
- **Cursor pagination:** Client sends a custom marker *(e.g. composite index from id and timestamp)*, DB uses `WHERE` clause to jump ***directly*** to the correct position. This has ***way better performance*** and scales consistently with dataset size. It is also ***stable under changes*** because it follows a logical record and not a numeric position. But the marker needs to have a DB index to actually improve performance.

Implementation requirements

- ***Opaque tokens:*** hide internal structure, prevent tampering *(don't use your primary key)*
- ***Composite index*** on sort fields *(often `created_at` + `id`)*
- ***Stable sort order:*** must be deterministic/unique
- ***Cursor expiration:*** prevent stale state exploitation

GraphQL: has a built-in cursor pagination via Relay Connection spec. Uses edges, pageInfo, first/after/last/before.

Critical pitfalls

- ***Missing/wrong index:*** Fallback to full table scan with $O(n)$ queries
- ***Non-unique sort keys:*** Leads to inconsistent ordering/results
- ***Poor index design:*** Can be worse than no index

Tradeoffs

- Cannot ***jump*** to ***arbitrary*** pages
- Higher implementation ***complexity***
- ***Multiple indexes*** for multiple sort options

For ***small/static datasets***, internal tools where ***simplicity*** matters or if you need ***page numbers*** or ***random access***, ***offset*** is the ***better choice***. The need for random access can be minimized by providing ample filter criteria.

13.5. REQUEST BUNDLE / BATCH OPERATIONS

Batching combines ***multiple operations*** into a ***single HTTP call*** to ***reduce*** network roundtrips, latency, and connection overhead. This is especially valuable for ***mobile clients*** with high latency or unstable connections.

Execution Models

- **Atomic batch:** Behaves like a transaction. If ***one*** operation ***fails***, ***everything*** is ***rolled back***. This ***ensures consistency*** but requires ***transactional backend support***.
- **Non-atomic batch:** Each operation is ***independent***. Return per-operation status in the response body. Use `207 Multi-Status` for mixed outcomes *(exact failures in body)*, `400` for a malformed batch input.

Critical constraints

- **Size limits:** Enforce max operations per batch to ***prevent DoS***. ***Document*** limits and require clients to split large batches.
- **Complexity limits:** Keep batches flat if possible, ***avoid nested/recursive dependencies*** to ***conserve*** server resources. Ensures ***bounded execution***.
- **Transaction boundaries:** Avoid cross-resource batches that span databases, ***limit scope*** to a single data store. ***Prevents*** distributed transaction ***complexity***.

Error handling

Return ***structured aggregated*** errors as an ***array*** of ***per-operation errors*** instead of generic 500s. This enables ***client-side recovery logic*** and aligns with RFC 7807 error patterns.

13.6. LONG RUNNING REQUESTS & EVENT-DRIVEN API

Important concept for operations that take ***longer than a few seconds***. Should not block HTTP connections, instead ***return a `202 Accepted` immediately*** and then ***process*** the request ***asynchronously***.

Return a job ID in the response body and in the location header so the client can poll the job status or cancel it *(e.g. `GET /jobs/{id}` with status: pending/processing/succeeded/failed)*. A poll response may include a percentage and estimated time remaining to use for example in progress bars.

Polling is ***simpler*** to implement than push, no firewall/NAT issues. Client controls request timing, no server infrastructure for webhooks needed. ***Disadvantage:*** Adds constant overhead because of repeated requests.
Push reduces polling overhead. There are three options:

- ***Webhooks:*** Efficient for real-time events. Server pushes updates immediately, but requires openly accessible endpoint *(Not possible with a firewall)*
- ***WebSockets:*** Bidirectional real-time communication, long-lived connections. Best for frequent bidirectional updates, but needs more complex infrastructure.
- ***Server-Sent Events (SSE):*** Simpler than WebSockets for server → client push *(but no support for client → server)*. Uses standard HTTP, easier to implement than WebSockets.

Webhook delivery (push model)

- ***Registration:*** Done by client. `POST /webhooks` with url, `event_types`, `secret` *(string for HMAC key)*.
- ***Verification:*** Done by server. Server calculates HMAC-SHA256 signature in `X-Signature` header. Client needs to validate signature before processing.

Retry policy: Exponential backoff *(1s, 2s, 4s, 8s, 15m, 30m)*. After repeated failures disable webhook and notify client.

Result & completion: Webhook push with signed payload for real-time notification. Ensures delivery despite client downtime.

Result retrieval: With `GET /jobs/{id}/result`. Expires after 24–72h *(TTL enforced)*. This limits storage cost. Cancellation is idempotent *(`POST /jobs/{id}/cancel`)*.

Developer experience: You should provide a `/webhooks/test` endpoint with sample payload and signature.

13.7. BACKEND FOR FRONTEND (BFF)

Create ***client-specific adapter:*** Tailor API response for mobile, web or TV – ***no overfetching***. Optimizes payload for device constraints, reduces bandwidth and parsing cost.

Aggregation layer: ***Merge*** data from ***3+ microservices*** into a ***single response***. This ***reduces*** client ***roundtrips***, ***minimizes*** network calls and ***improves*** perceived performance.

Client-specific optimizations: Minimize payload size for mobile, prefetch related data, support offline caching. ***Essential for low-bandwidth environment.***

- ***GraphQL BFF:*** Expose different schema per client. This allows field selection and avoids versioning churn.
- ***REST BFF:*** Version endpoints via `/v2/mobile/users`, avoid breaking existing clients.

Multi-Platform management: Isolate BFF per platform for versioning. Enables ***independent release cycles.***

Security: BFF is a ***trusted client*** – authentication happens in BFF, calls from then on happen with service accounts.

13.8. API LIFECYCLE MANAGEMENT

Ensures that APIs can be systematically ***updated without affecting*** existing clients.

There are multiple ways to introduce new changes:

- ***Feature flags:*** Enable new behavior for internal/test clients before public release. Gradual rollout.
- ***Experimental endpoints:*** Mark with `/v1/experimental/` – no SLA, subject to removal. This clearly identifies unstable endpoints.
- ***Beta/Alpha labeling:*** Use `/v1/beta/users` or `Accept:application/vnd.myapi.v1+beta`. Also clearly identifies unstable endpoints.

Deprecation Process

Announce deprecation via email, dashboard and sunset header *`(Sunset: Wed, 31 Dec 2025 23:59:59 GMT; rel="deprecation")`* at least 90 days in advance. ***Monitor*** usage of deprecated endpoints and alert when more than 1% of the traffic remains.

Enforcement: Block access to sunset endpoints using TTL – return a `410 Gone`.

There is always ***risk of poor management:*** People ignore deprecation which leads to sudden outages, compliance risks, customer churn.