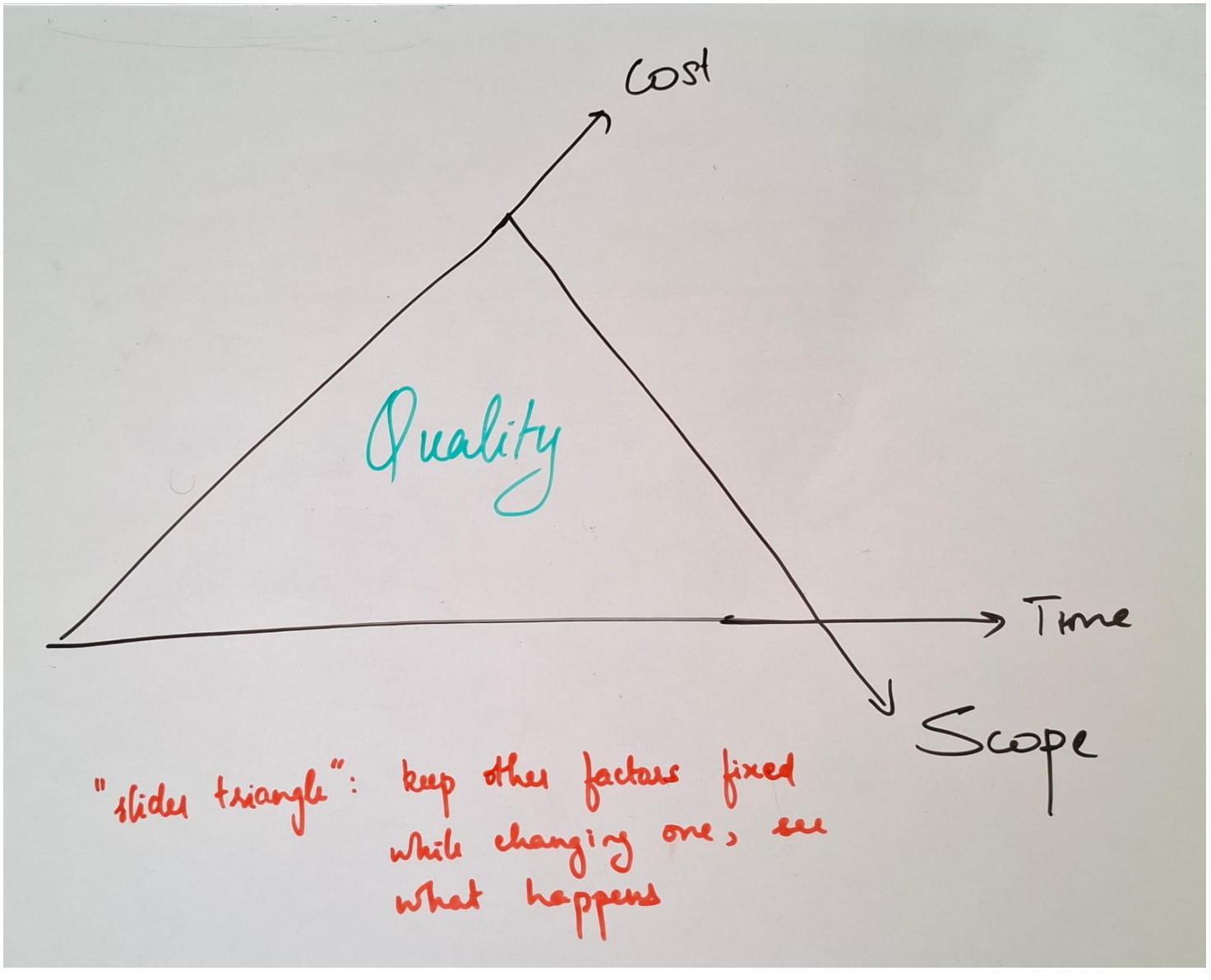
Unit 2

Project Management

Software Project Manager

- Leader
  - Clearly make everyone understand their roles and what is expected of them
  - Give them overall direction
  - Build a team comprising of people with different skills
  - Monitor course of progress
- Liaison
  - Link between team, clients and higher ups
  - Work with analysts, designers and other teams as required
  - Update all stakeholders on progress as needed
- Mentor
  - Be there to guide team at every step as needed

Project Quality



Agile Planning

- Planning for next functionalities to be implemented done in incremental steps
- Direction of project decided during development, not prior
- Allows for flexible plans that change with changing customer requirements and priorities

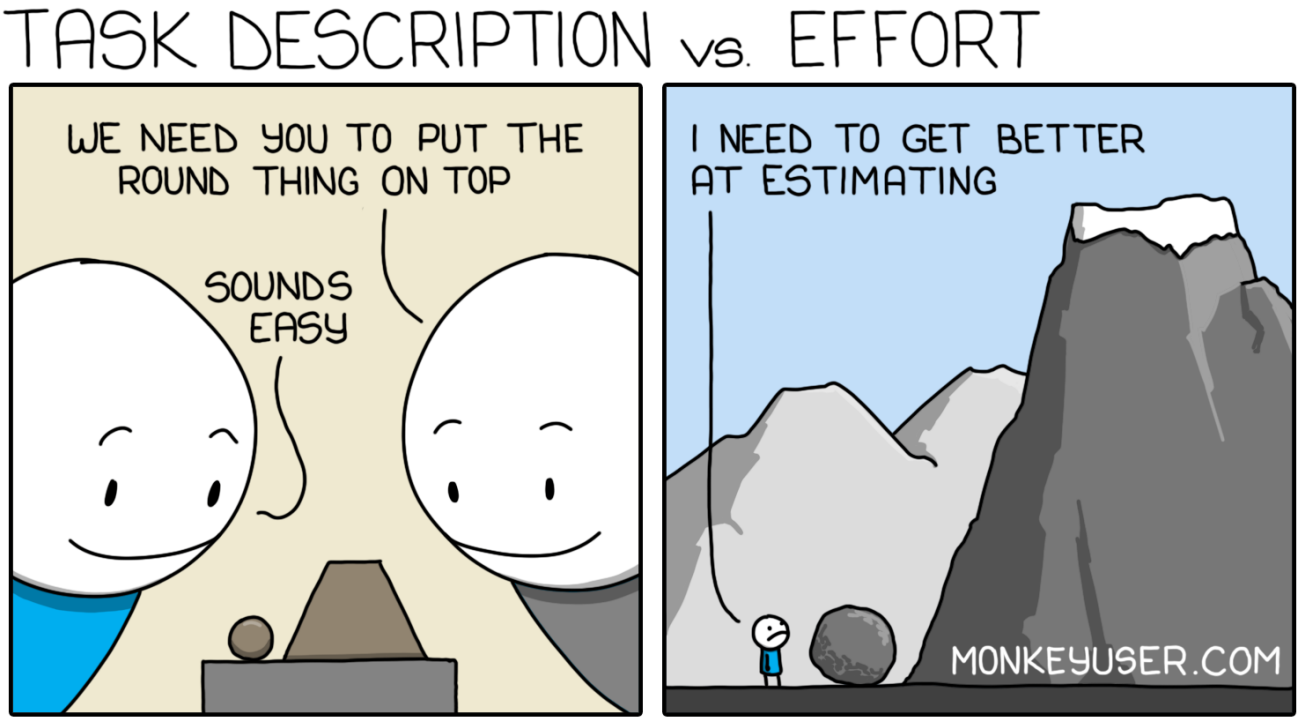
The Process: Planning Game

1. Story Identification

- Identify features required by the user as user stories
- "As a <user>, I need <feature> because <reason>"

2. Initial Estimation

- Decide the amount of "effort points" for each story based on difficulty and amount of time needed to implement them
- Number of "effort points" implemented by the team on average per day = velocity of the team
- This allows for the estimation of the overall effort required to be put in by the team and the amount of time it will take to complete tasks



3. Release Planning

- Long term plan
- What functionalities will be included in the next release version of the product?

- Order functionalities based on customer priority to pick

4. Iteration Planning

- Short term plan: 2-3 weeks
- Stories to be implemented in that iteration; number of stories depending on the effort points assigned to them
- Velocity is key here to pick the right number of stories

5. Task Planning

- Immediate plan
- Break down stories (if needed) into development tasks; each task not more than 16 hours of dev at maximum
- Devs on team take ownership of separate development tasks

Delivery

- The release or iteration schedule is never pushed. New software is always delivered at the end of every iteration
- If a mistake has been made in allocation, then that is remedied by reducing the scope as necessary for a given iteration to ensure quality

Agile Planning Difficulties

- Requires customer to be in the loop to know what their current priorities and requirements are
- Many customers may not be available to participate the entire planning game because they have better things to do with their life
- Customers may also be used to traditional project plans and not this iterative approach

When is it applicable?

- Small, stable dev teams where everyone is able to meet regularly to discuss progress and next steps

---

Time Estimation

- Done using a **Work Breakdown Structure (WBS)**
- Overall description of how work will be done for a given project

Constructive Cost Model - CoCoMo

- Regression model based off of Lines of Code (LOC)
- Cost estimate model used to predict the amount of effort, time, cost etc
- Key factors that are an outcome:
    - **Effort:** How much labour is required to complete a given task. Usually measured in person-month units

- **Schedule/Time:** Amount of time needed to complete the project, proportional to effort

- Types of CoCoMo:
  - Basic
  - Intermediate
  - Detailed

> **Note**
> **Types of Projects wrt Constants to Be Used In CoCoMo**
>
> 1. Organic:
>   - Small team size required
>   - Well understood problem that has been done before
>   - Team has nominal experience with the problem
>   - Eg: Inventory management system
> 2. Semi-detached:
>   - Everything in between organic and embedded
>   - Eg: Building an OS
> 3. Embedded:
>   - Highest level of everything
>   - Larger team
>   - Experienced, creative devs
>   - Eg: Large, at-scale solutions like creating the software for the entire ATM network of a bank

Basic CoCoMo

$$Effort\ (E) = a(KLOC)^b$$

where KLOC = Kilo Lines Of Code

$$Time\ (t) = c(Effort)^d$$

And finally:

$$Persons\ Required = \frac{Time}{Effort}$$

Software Project	a	b	c	d
Organic	2.4	1.05	2.5	0.38
Semi-detached	3.0	1.12	2.5	0.35
Embedded	3.6	1.20	2.5	0.32

Risk Management

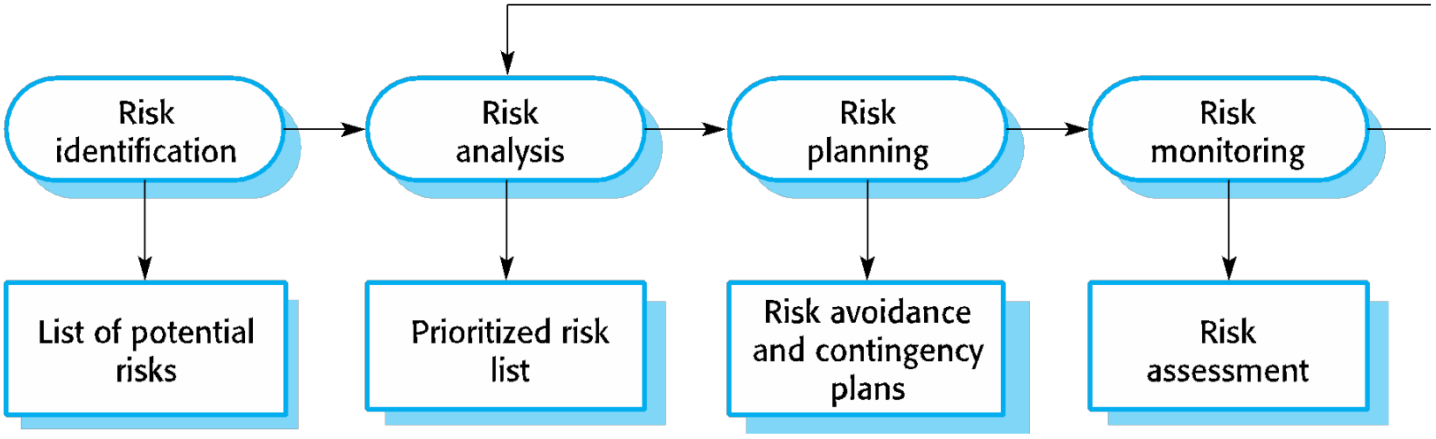
Risk Classification

- By type:
  - Organisational, technical etc.
- By effect:
  - Project: Affects schedule or resources
  - Product: Affects functionality, performance
  - Business: Affects org making software

Risk	Affects	Description
Staff turnover	Project	Experienced staff will leave the project before it is finished.
Management change	Project	There will be a change of organizational management with different priorities.
Hardware unavailability	Project	Hardware that is essential for the project will not be delivered on schedule.
Requirements change	Project and product	There will be a larger number of changes to the requirements than anticipated.
Specification delays	Project and product	Specifications of essential interfaces are not available on schedule.
Size underestimate	Project and product	The size of the system has been underestimated.
CASE tool underperformance	Product	CASE tools, which support the project, do not perform as anticipated.
Technology change	Business	The underlying technology on which the system is built is superseded by new technology.
Product competition	Business	A competitive product is marketed before the system is completed.

Risk Management Process

Identification -> Analysis -> Planning -> Monitoring



Risk Indicators

Risk Type	Indicators
Estimation	Failure to meet original schedule, frequent delays
Organisational	Lack of action from senior management, org gossip
Technical	Many reported technical issues, late delivery of hardware/support software
Tools	Reluctance of team to use assigned tools, complaints about CASE tools being used (Computer-Aided Software Engineering tools)
People	High turnover, poor relationships amongst team members
Requirements	Many requirement change requests, customer complaints about product not matching expectations

---

Roles - RACI Matrix

- **Responsible:** The doer(s) responsible for a given action or task. Can have multiple Rs for a single task
- **Accountable:** Buck stops here. Only one A per given task or activity. Can veto decisions.
- **Consulted:** People to talk to before making major policy changes like experts, higher ups. Two way comms.
- **Informed:** People who just need to know current status. One way communication.

Task & Stakeholders	CIO	CTO	CEO	Project Managers	Sales Team	Marketing Team	Others
Define feature requirements and specs	C	I	R	A	I	C	C
Develop the new feature	C	R	I	A	I	C	C
Test the new feature	C	C	I	R	I	A	C
Create marketing materials and campaigns	C	I	R	C	I	A	C
Train sales team on the new feature	C	I	R	C	A	I	C
Coordinate with customer support	C	I	R	C	C	I	C
Monitor and analyze user feedback	C	C	R	A	I	I	C

Project Monitoring and Control

Monitoring: See what is happening
Control: Modify cost, resources allocated, scope, time, organisational structure as needed based on above

Gantt Chart, Critical Paths

- Critical path: longest path sequence of interdependent tasks that needs to be completed to finish the project
- Activities that have slack time of zero $\in$ critical path
- Dependencies: Tasks that must be completed before others can start.
- Float (Slack): The amount of time a non-critical task can be delayed without affecting the project's end date.
- Critical Tasks: Tasks on the critical path with zero float; any delay here delays the project.

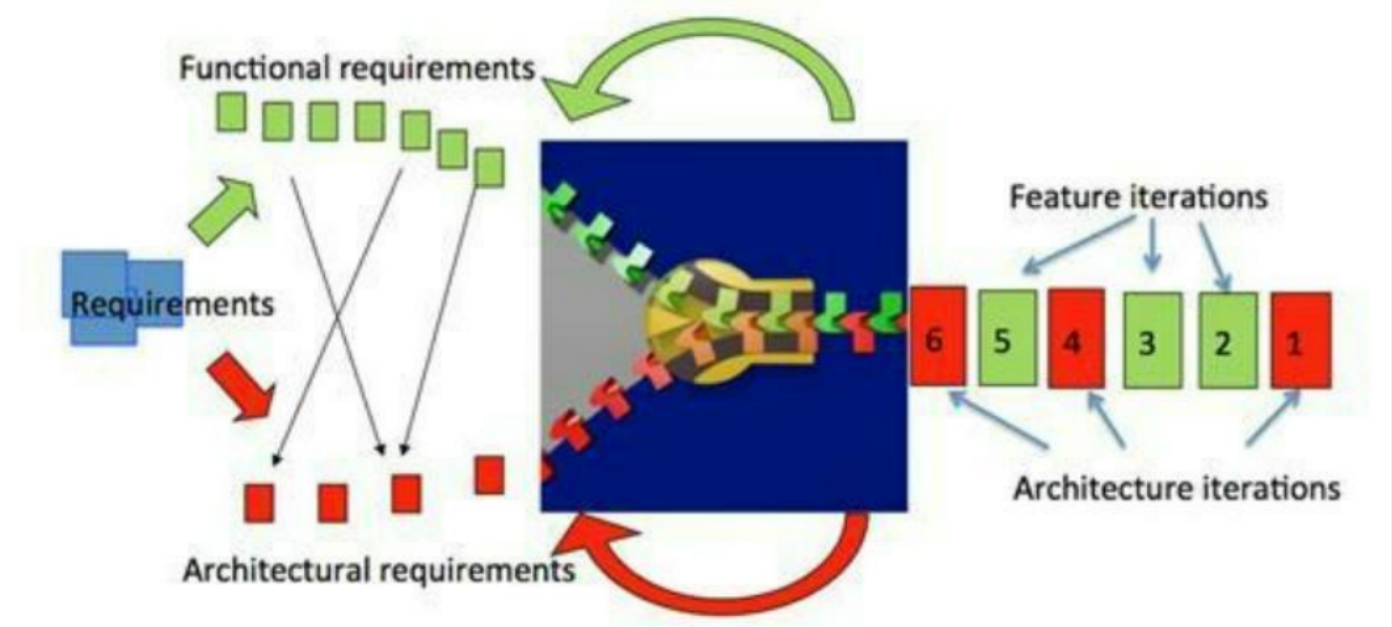
Critical Path Analysis

1. List Activities: Identify all tasks needed for the project.
2. Define Dependencies: Determine the order; which tasks must finish before others start (predecessors).
3. Estimate Durations: Assign a time estimate for each task.
4. Create a Network Diagram: Visualize tasks and dependencies (often using Activity-on-Node).
5. Calculate Paths: Find all possible paths from start to finish, summing task durations for each.
6. Identify the Critical Path: The longest path is the critical path, defining the project's shortest time.

Software Architecture

- Top level decomposition of project into major components and modelling of how these components will interact
- Mid level what, high level how
- Every system has arch, whether consciously decided or not
- Arch decisions influenced by technical, business, social and non-functional requirements

Design Questions	Architectural Questions
How do I add a menu item in the app?	How do I extend app with a plugin ?
How can I make it easy to use a function?	What threads exist and how do they co-ordinate?
What lock protects this data?	How does Google scale to billions of hits per day ?
How does Google Rank pages?	Where should I put my firewalls ?
What encoder should I use for secure communication ?	What is the interface between subsystem ?

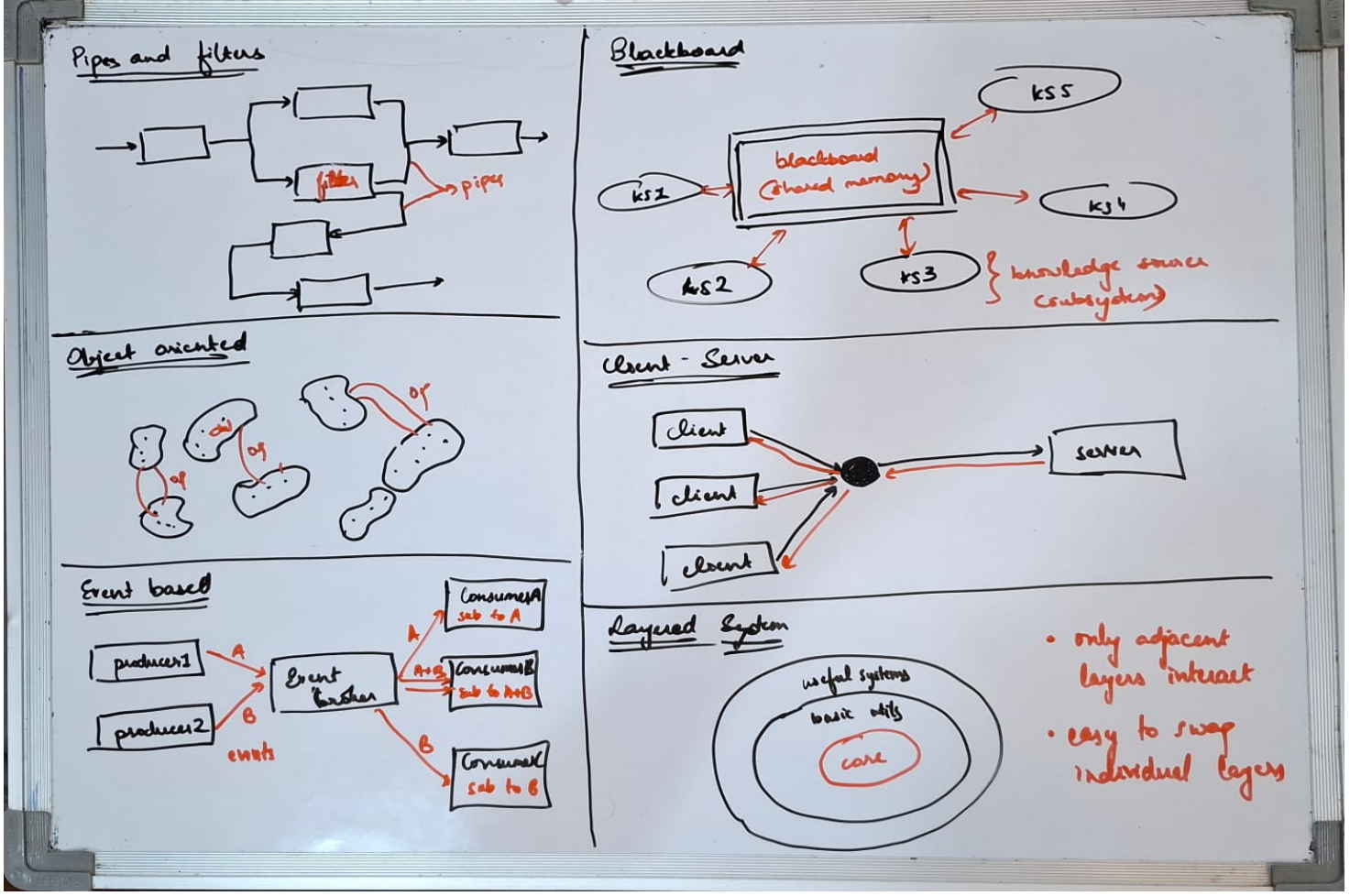
> **Note**
>
> **Agile Architecture Decisions - The Zipper Model**
>
> 
>

Architectural Patterns and Styles

Pattern: General, reusable solution used to solve a common problem that comes up in software arch within a given context
Style: Repeatable sequence of arch decisions taken that:
- are applicable in a given context,
- enforce certain constraints on the decisions,
- and elicit beneficial qualities in that given context.

> Pattern needs a problem to exist, style is more general.

Types of Patterns/Styles



- Pipes compute local transformations, filters must not share states, no cycles. Eg: Compilers
- Encapsulation, inheritance, polymorphism. Complex data models, code reusability. Eg: Adobe Photoshop
- Asynchronous, loosely-coupled components. Eg: Node.js
- Subsystem (knowledge source) directly accesses shared main mem and contributes partial solutions. Eg: Distributed systems, stock exchange
- Eg: Online banking system
- Eg: Internet Protocol Suite, OS

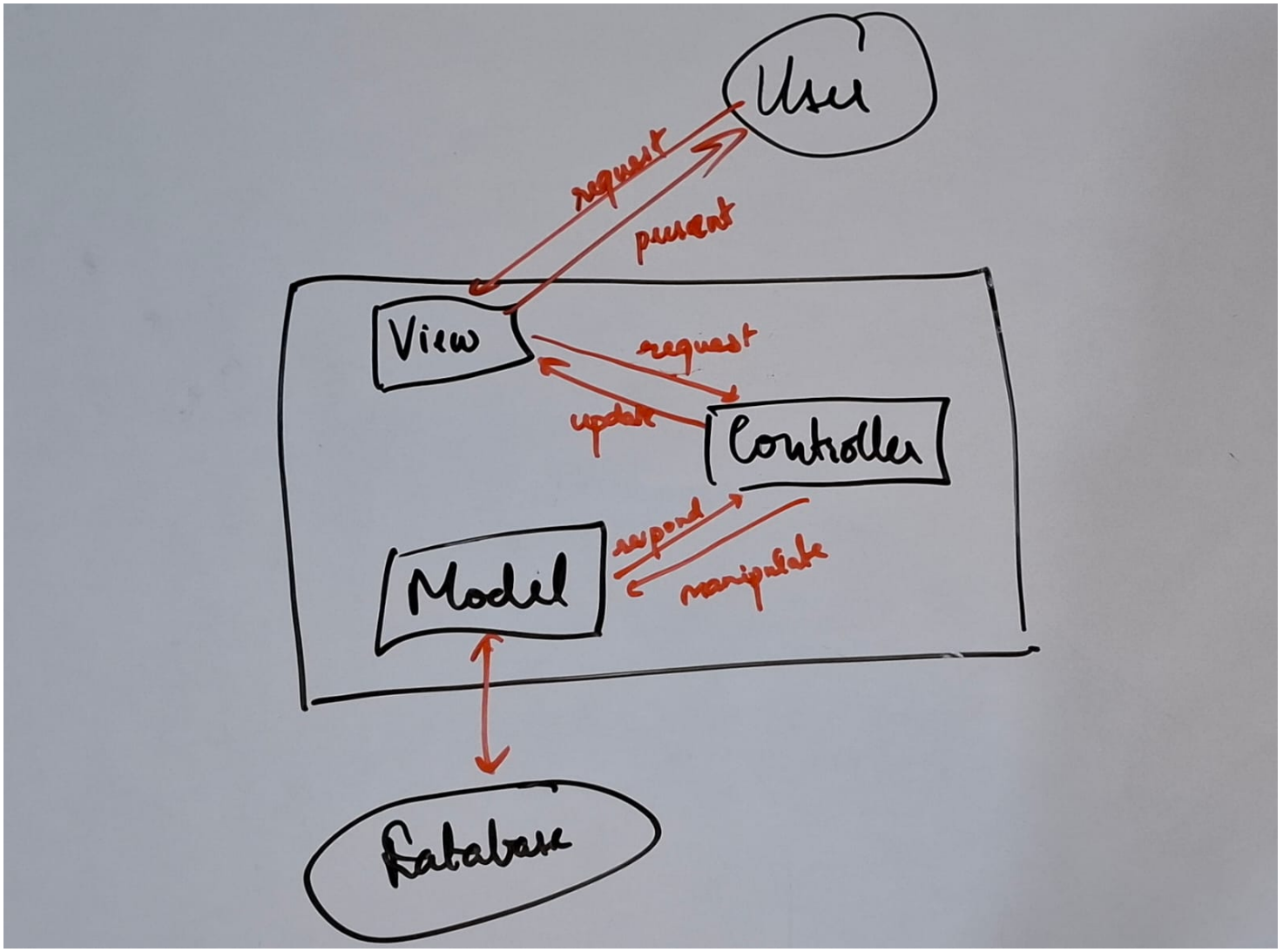
Considerations

- Level of abstraction: What the components are, how they communicate with each other
- Separation of concerns:
  - **Cohesion:** Cohesion is high if the operations within a module are highly related to one another -> clear scope. Good abstraction = strong internal cohesion.
- Modularity:
  - **Coupling:** How easy it is to work with components individually, independent of other components. High coupling = harder to work with components individually = harder to make changes or debug

Model View Controller (MVC) Model

Model: The internal business logic and application. Core functionality.
View: Interface presented to user

Controller: Handles control flow; acts as a mediator between the view and model layers



Security Design and Architecture

Do arch but put a lil security in there

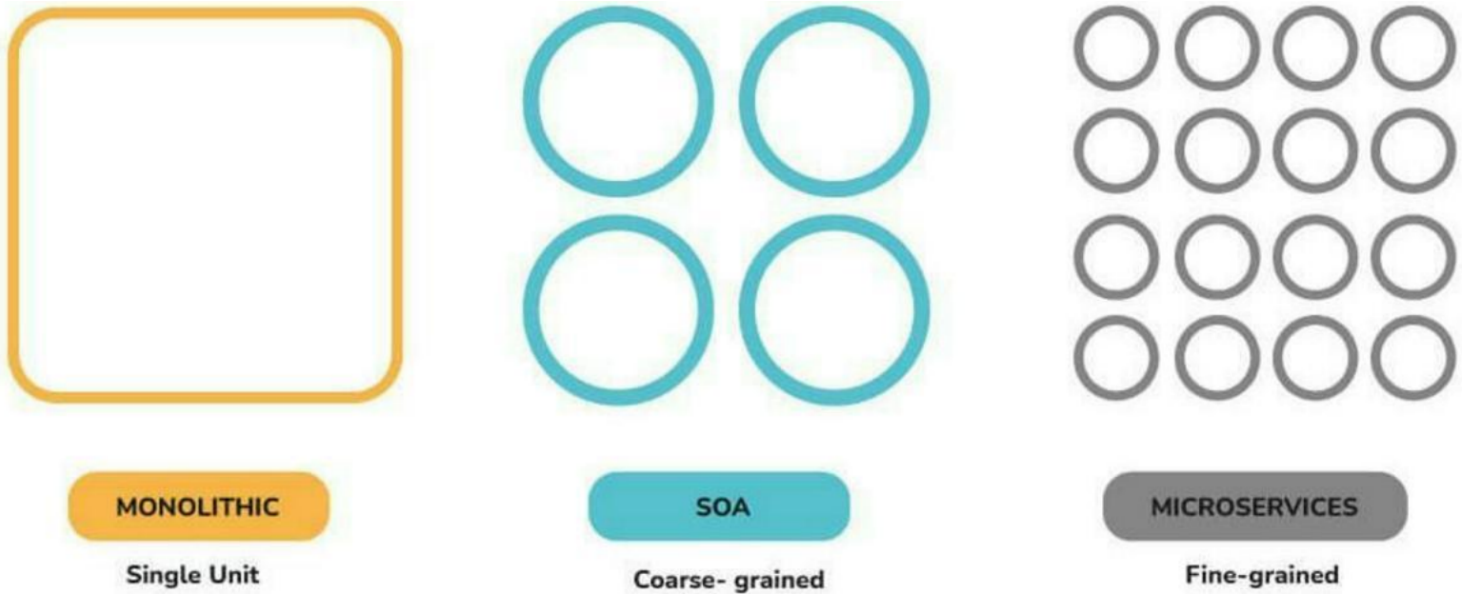
- Reduce attack surface
- Increase defense depth
- Ensure separation of concerns
- Align with security requirements

Security Design Patterns

- Input validation layers
- Secure modes of comms (TLS, mTLS)
- Encapsulation and service boundaries
- Create trust boundaries for subsystems
  - Components within trust boundaries -> mutually trusted
  - Define who has access to each trust boundary and how they have access
- Consider assets at all points: origin, transit, rest, use, discard

Microservices

Multiple smaller services make up the overall functionality; communicate through API requests.

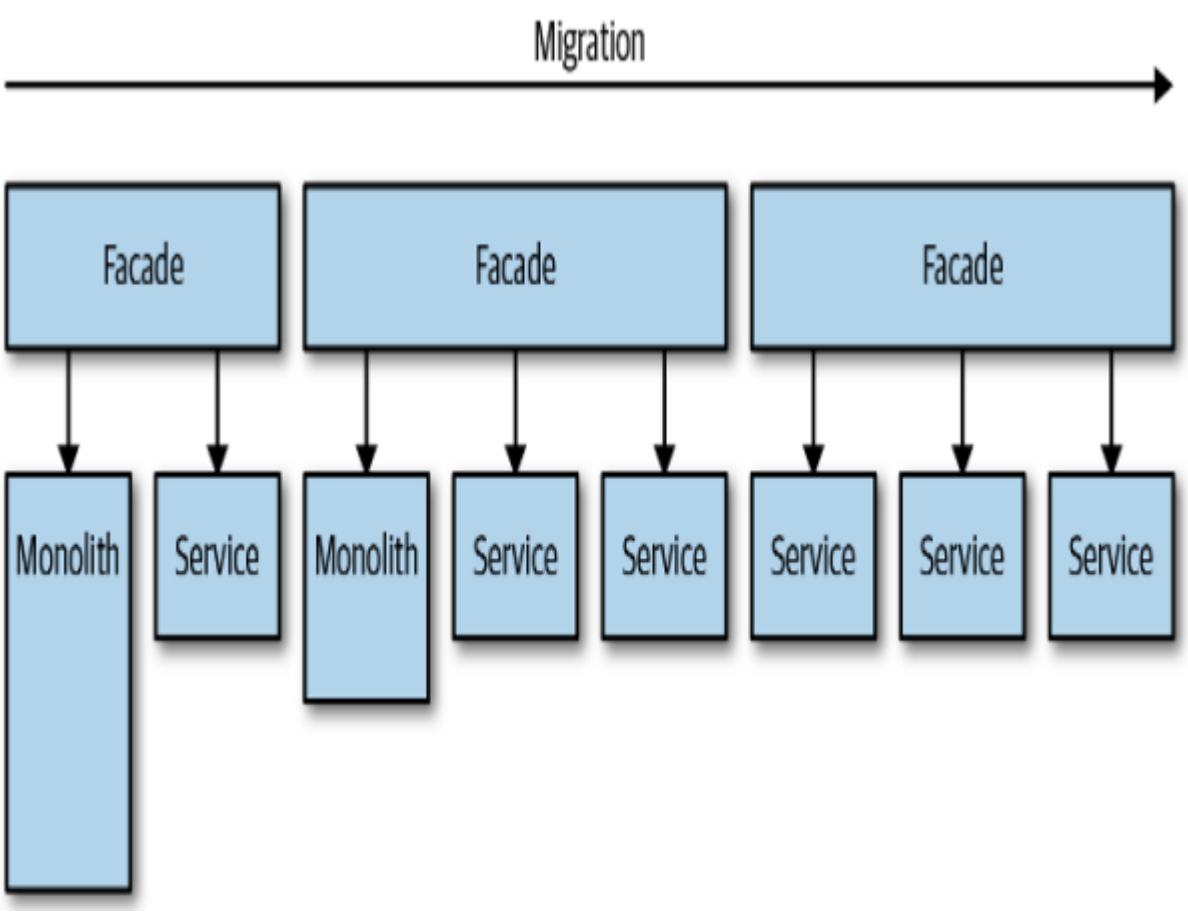


- Ship features faster and safer
- Update certain services or components more frequently than others
- Scalability
- Fault tolerance
- Architecture can handle different kinds of computational demands

Monolith To Microservices

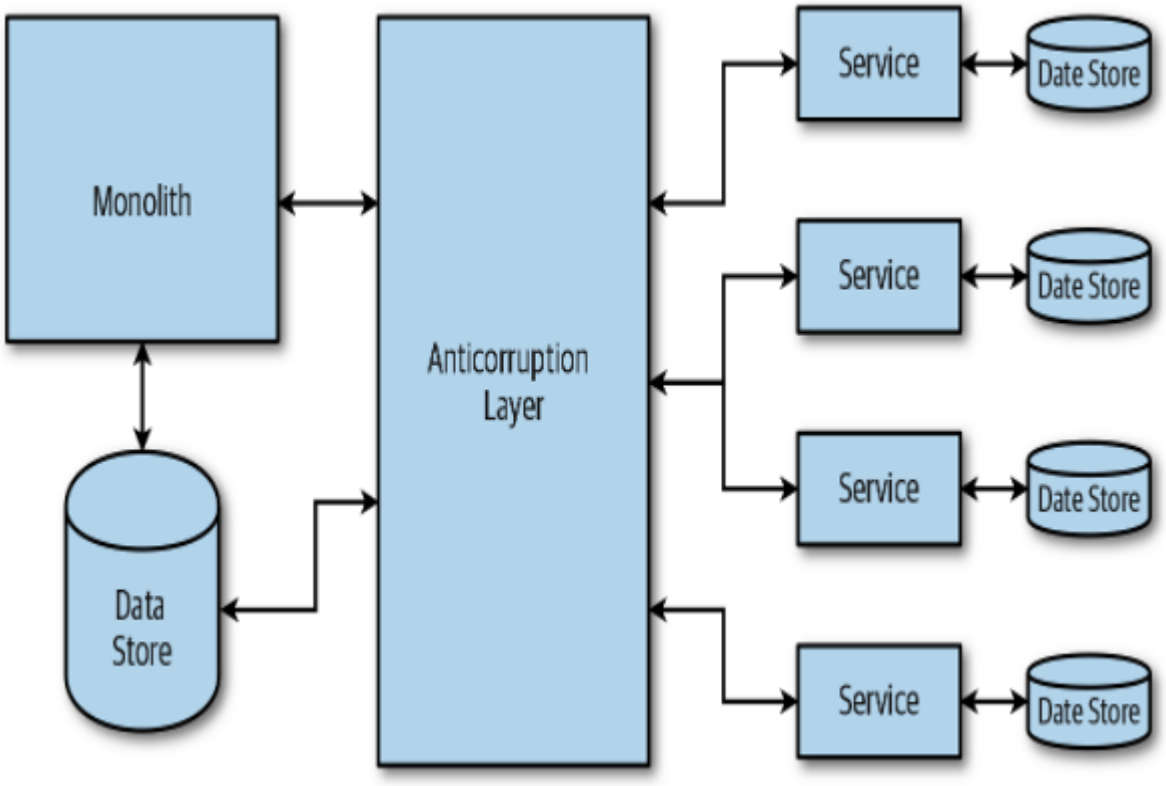
1. Strangling

- New services and existing components implemented as microservices one by one
- Facade/gateway routes the user to the right service (monolith/microservice)
- Over time, replace everything with microservice -> remove facades -> strangle or retire legacy monolith



2. Anticorruption Layer

- Used when microservices need to access legacy monolith
- Implement a facade between between components that do not share the same semantics
- Anticorruption layer -> translate between different protocols/schema



Why NOT Microservices?

Increased Operational Complexity - Managing and deploying multiple services
Example:
In a fintech platform, managing independent microservices for payments, authentication, user management, and analytics requires maintaining separate deployment pipelines and health checks. Any misconfiguration in one service (e.g., wrong environment variable) could cause cascading failures during deployment

Inter-Service Communication - Ensuring reliable communication between services
Example:
A microservice for order processing calls an inventory service to check stock. If the inventory service is slow or down, the order service times out. To mitigate this, developers implement retries with exponential backoff, but improper configuration can cause a "retry storm," worsening the load on the already degraded service.

Data Consistency and Integrity - Managing transactions across distributed services
Example:
In an e-commerce platform, placing an order involves updating the order service, inventory, and payment services. Using a Saga pattern, each service commits locally and compensating transactions are used to roll back if something fails. If the payment service fails after inventory is deducted, the inventory must be rolled back separately.

Testing and Debugging - End-to-end testing is hard when services depend on each other
Example:
Testing a customer onboarding workflow requires running services like KYC validation, email notification, user provisioning, and analytics. Mocking each service's responses is non-trivial, and bugs often emerge only in staging or production due to real service behavior not matching mocks

Governance and Standardization - Lack of uniform standards across services
Example:
One team uses REST with camelCase JSON keys (`userId`), while another uses snake_case (`user_id`). When integrating services, the inconsistencies cause serialization/deserialization issues, leading to production bugs and friction during integration.

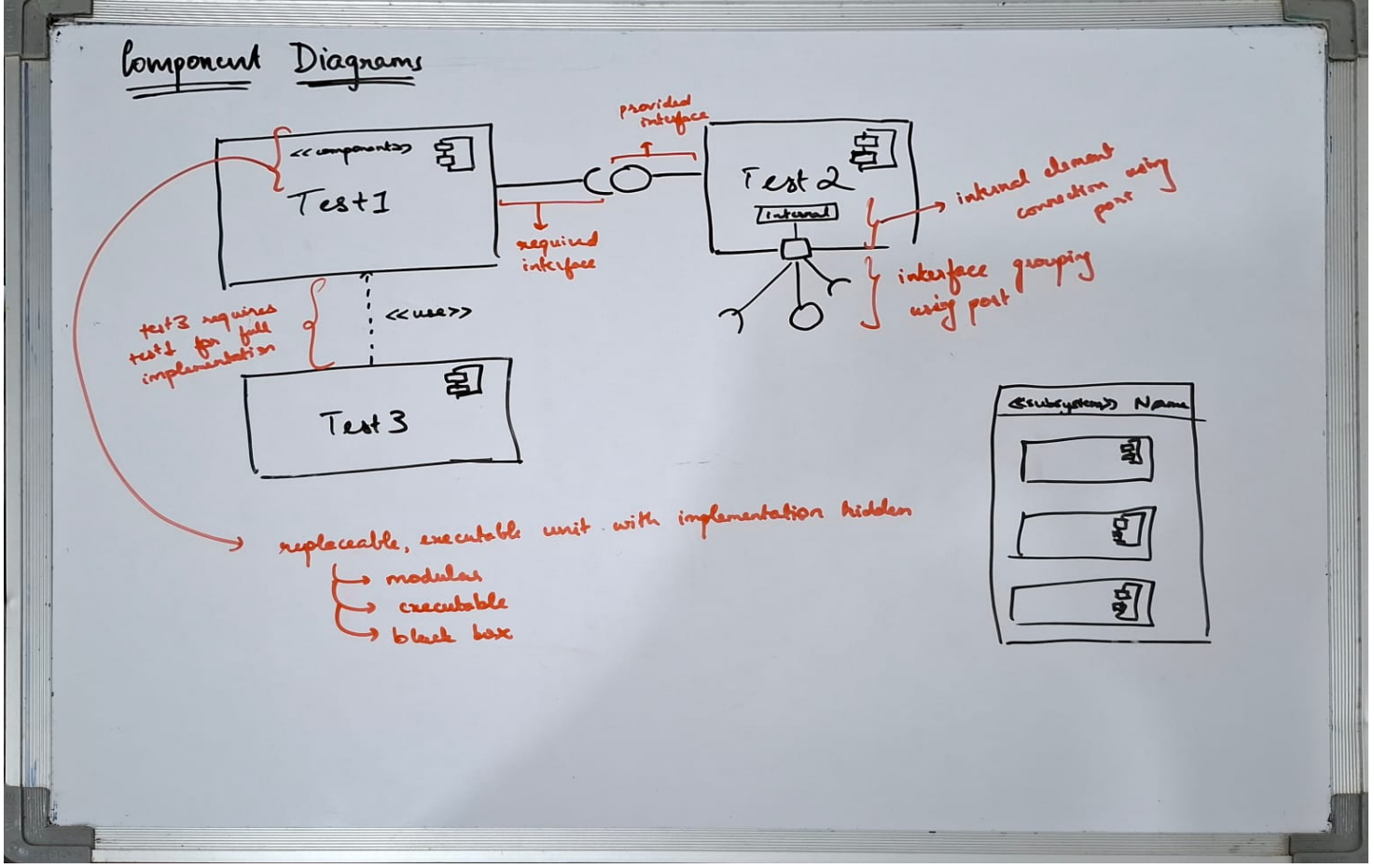
DevOps and Infrastructure Overhead - Managing infrastructure for each service
Example:
Each microservice (e.g., notifications, audit logging, billing) requires its own Dockerfile, Helm charts, CI/CD pipeline, monitoring dashboards, and alert rules. The operations team spends more time on infrastructure maintenance than on improving product capabilities

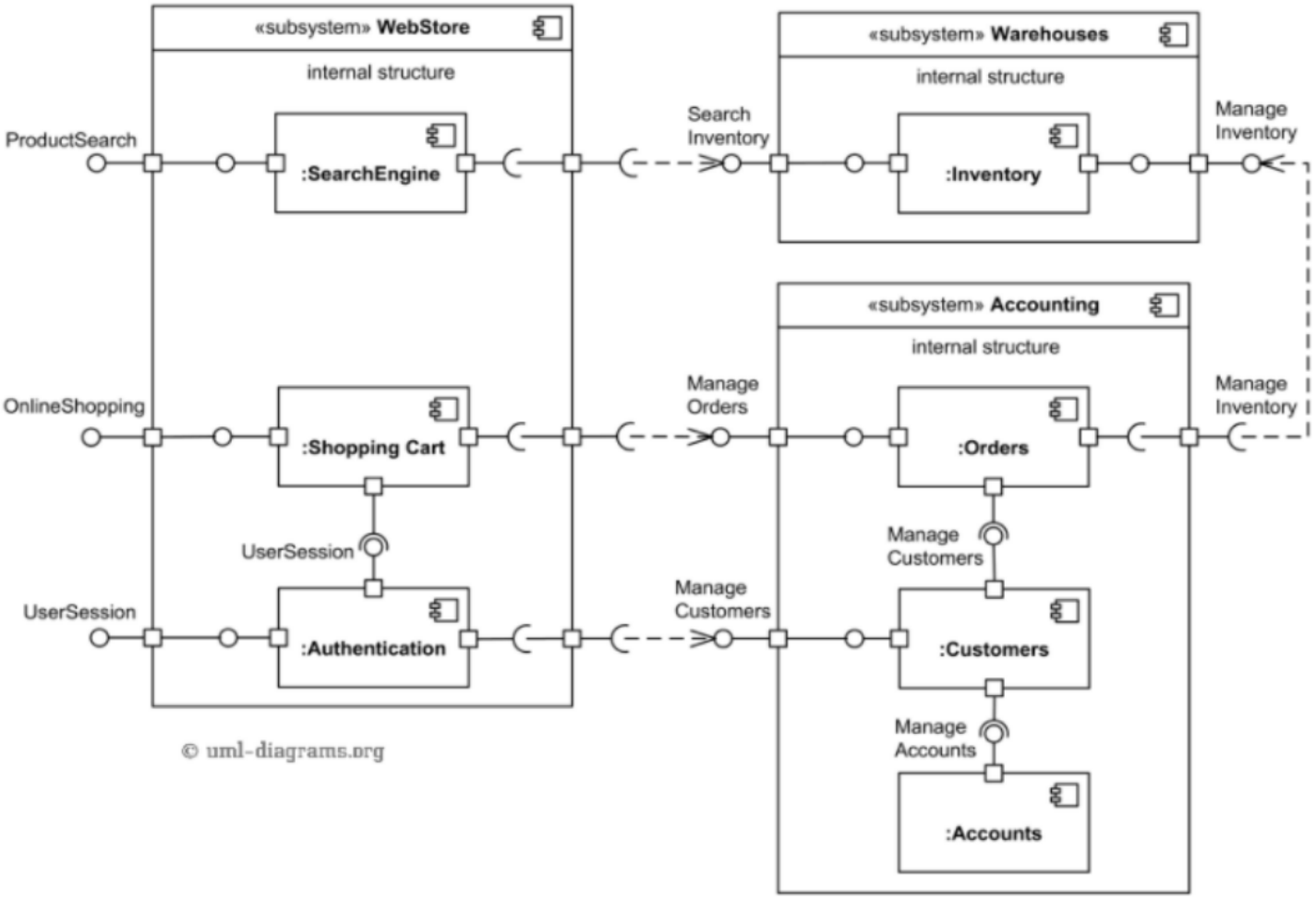
Team and Organizational Challenges - Coordination and skill gaps between teams
Example:
A team responsible for the billing microservice decides to upgrade to a new database version. However, they fail to notify the data ingestion team, whose service depends on specific billing data formats. This causes downstream failures during deployment

Versioning and Backward Compatibility - Managing breaking changes across services
Example:
A user profile service renames a field from `fullName` to `name`. Several consuming services, including notifications and analytics, break because they were not updated in time. Even with versioned APIs, maintaining compatibility across consumers is difficult.

Security and Compliance - Securing communication and enforcing compliance across services
Example:
In a healthcare system, services like patient data, prescription management, and reporting must all comply with HIPAA. If the report-generation service accidentally logs sensitive patient data without encryption or access control, it can lead to a major compliance violation

Component Diagrams





Software Quality

Quality Attributes: FLURPS+

Functionality: Does it do what the user needs it to do correctly?
Localisation: Easily adapted to local languages
Usability: Easily usable for the users; good documentation
Reliability: No crashes; check frequency of failures in intended time
Performance: Speed, throughput, resource consumption
Supportability: Maintainability, serviceability etc.
+: Extensibility etc.

Terminology

Attribute: Measurable physical/abstract quantity of a given entity
Measure: Quantitative indication of something. Eg: Number of errors
Metric: Quantitative measure to which an entity possesses a given attribute; calculation between two measures.
Eg: Number of errors / KLoC

Characteristics of Measures and Metrics - QUAREL

Quantitative: Definite measurement possible
Understandable: Should be very intuitive to understand what the measure is doing and how it's being calculated
Applicability: Should be applicable across all stages of the SDLC
Repeatability: Repeated application of the metric on the same inputs should give the same outputs
Economical: Calculation of the metric should not be expensive
Language-Independent: Should not depend on programming language used in the project

Cost of Quality

Determine the amount of resources of the org that go into preventing poor quality, check quality of deliverables and dealing with the results of internal and external failures.

Cost of Good Quality	Cost of Bad Quality
Prevention Cost: Cost of preventing poor quality results Eg: Error proofing, improvement activities	Internal Failure Costs: Cost of repairing issues that were caught or found internally before the customer receives the product Eg: Rework, retesting
Appraisal Cost: Cost of checking quality right now; conformance to standards and requirements Eg: Quality Assurance, Inspection	External Failure Costs: Cost of repairing issues that were found after delivery to the customer Eg: Support calls, patches
Management Control Cost: Cost to prevent failures in management activities Eg: Contract reviews, release criteria	Technical debt: Cost of fixing a technical issue which if left unfixed, could cause issues in the software Eg: Structural problems, unnecessarily complex code
	Management Failure Costs: Costs incurred by the personnel due to poor quality software Eg: Unplanned costs, customer damages

Categorisation of Software Metrics

Direct

- Depends only on direct values
- Internal attribute
- Eg: Cost, Effort, LoC etc.

Indirect

- Derived from direct values
- Eg: Productivity, defect density etc.

Size Oriented

- Errors/KLoC, Cost/LoC etc.

Complexity Oriented

- Fan-In, Fan-Out:
  - Fan-in -> number of modules that reference a given module
  - Fan-out -> number of modules referenced by a given module
  - Ideal: High fan-in, low fan-out
- Halstead's measures of entropy
  - Treat code as a collection of operations and operands
  - Certain formulae to estimate complexity, how hard it is to understand, how many expected bugs etc.
- Program length, volume, vocabulary: determined from above

Project

- Productivity
- No. of devs
- Cost
- Schedule

Product

- Assess state of projects
- Assess risks
- Evaluate ability to control quality

Process

- Long term process improvements
- Insights from SE tasks

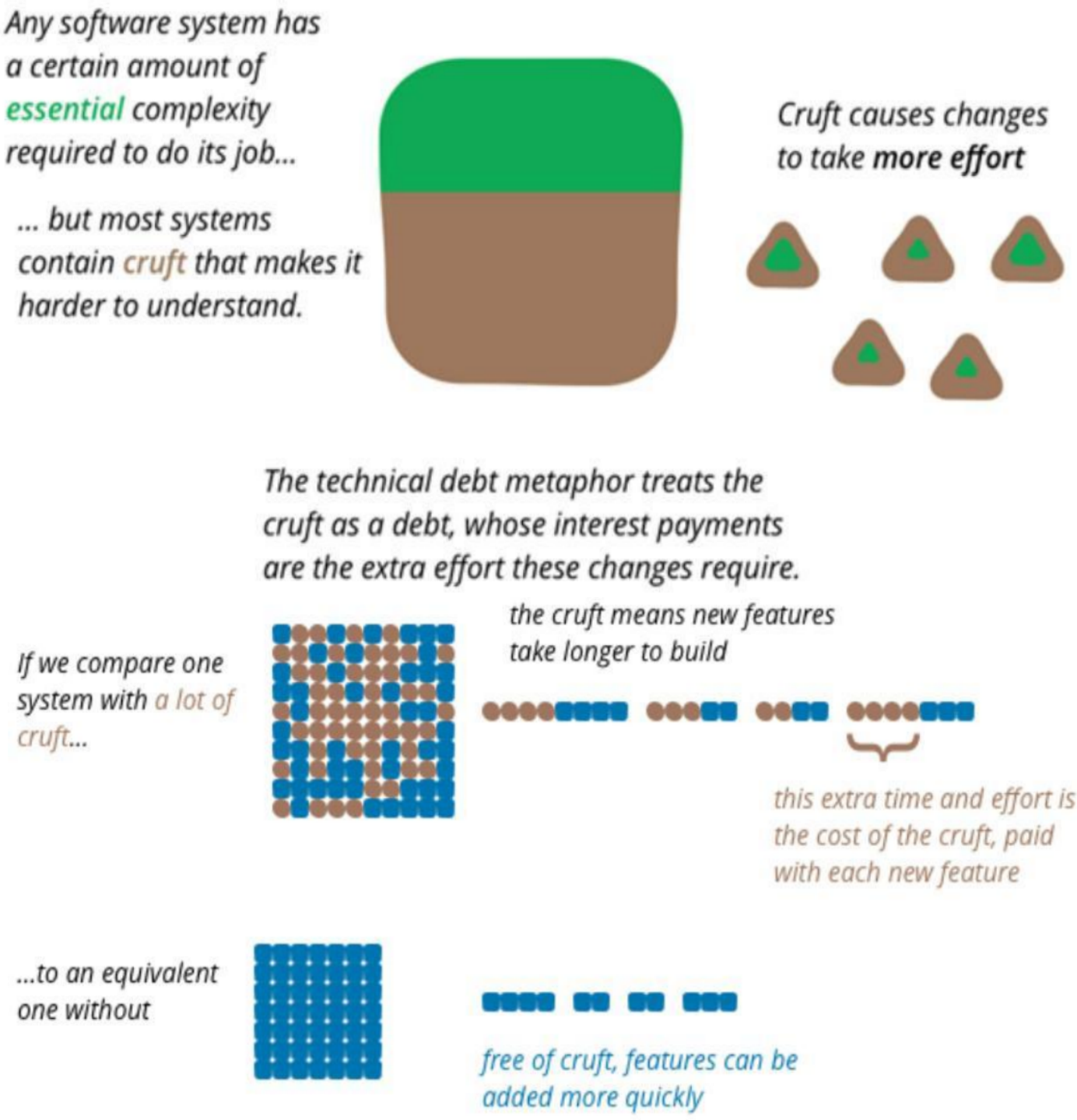
Software Quality Assurance

- Methods to monitor SE processes to ensure quality
- Quality has to be built in from the start, cannot be an add-on

---

Technical Debt

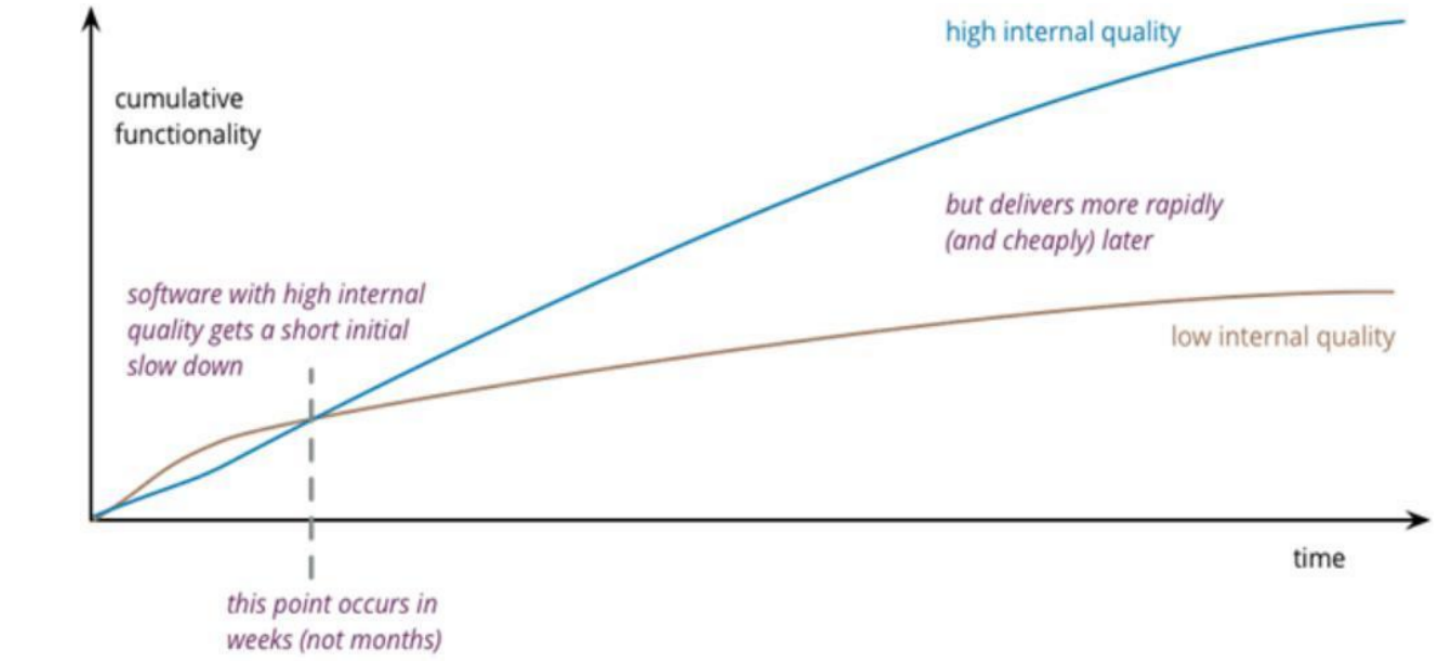
- Software systems = essential complexity + unnecessary complexity (cruft)
- Additional effort to add new features due to cruft = technical debt



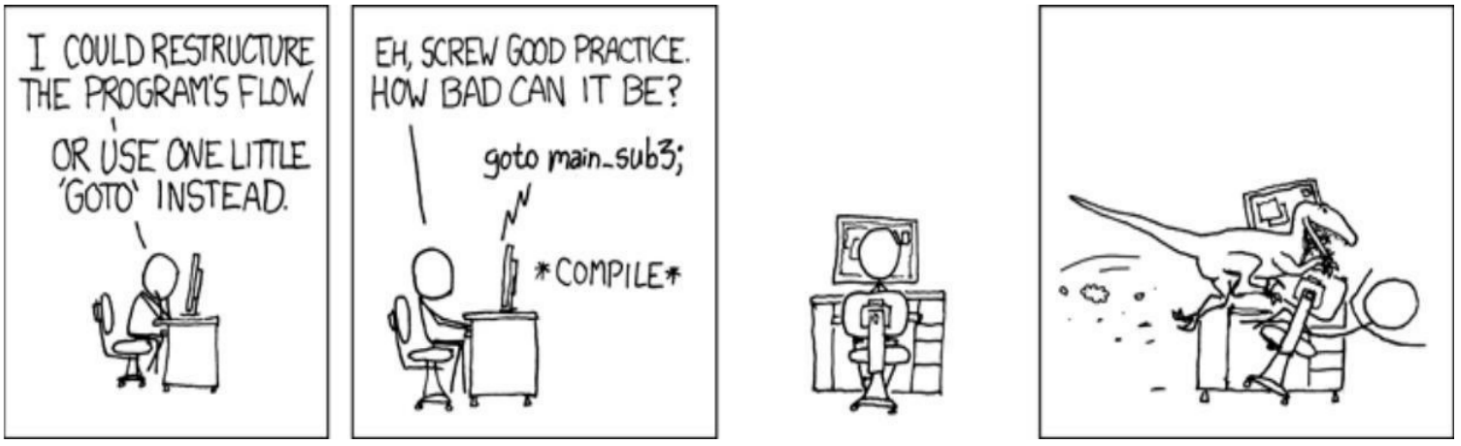
Technical debt is not necessarily bad internal quality!

- Design or implementation constructs that are expedient in the short term but make long term changes more expensive result in technical debt

- Technical debt impacts maintainability and evolvability

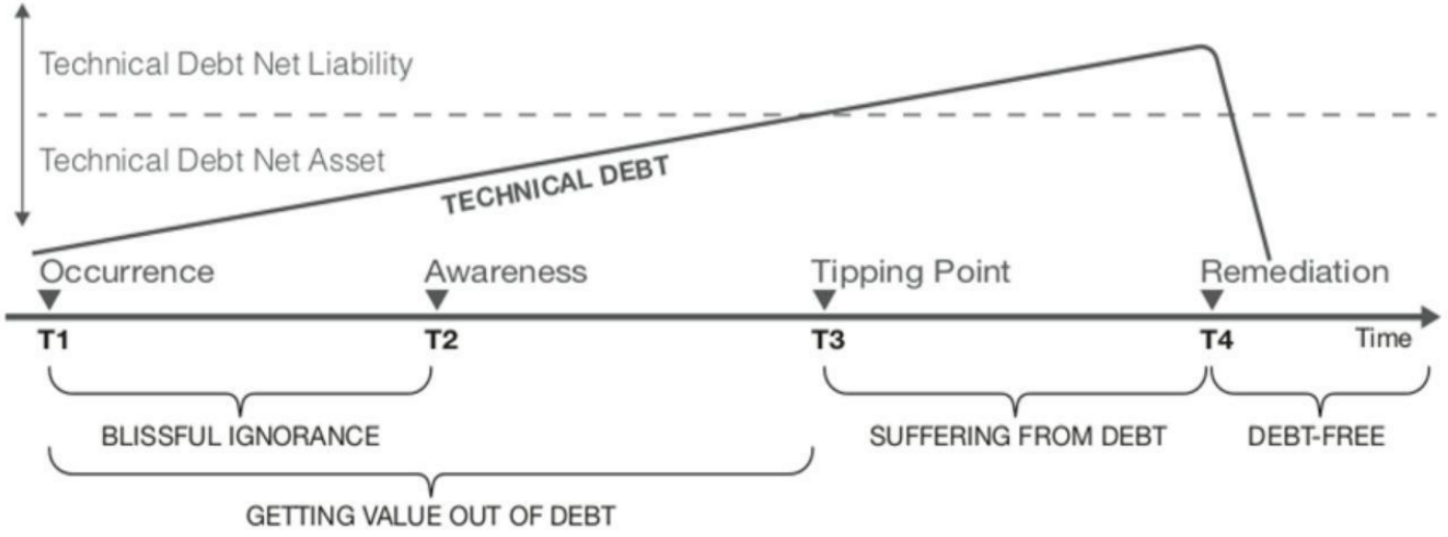


What causes technical debt?



Tightly coupled components, poor requirements, lack of process, lack of documentation, lack of tests, lack of ownership, delayed refactoring, long-lived dev branches etc etc etc.

Reducing technical debt



We need to first understand that there are different types of technical debt:

	Reckless	Prudent
Intentional	"We don't have time to set up a CI/CD pipeline"	"We must ship this now and deal with the consequences of not doing xyz later"
Unintentional	"What's a git pull?"	"Now we know we should not have done it"

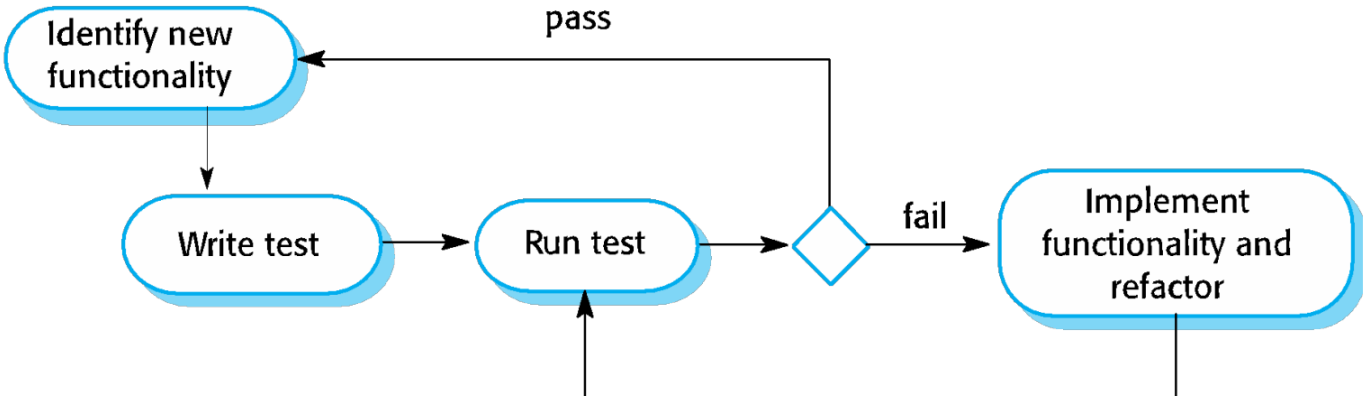
Intentional technical debt is dealt with pretty easily: just don't do it :D
Or fix it later, if it's really necessary to take a shortcut.

So how do we deal with unintentional technical debt? How does it happen?

- Not having QA processes or not following them
- Not using version control properly
- Slow and inefficient QA processes -> never get to develop new features
- Reliance on repeated manual labour for QA:
  - Superficial tasks
  - Mistakes will happen

Test Driven Development

- Identify new functionality
- Write unit test for it first and run all tests with the new one; the new one will fail
- Write code to implement said functionality that passes the test you have written for it
- Rerun all tests and ensure everything passes including the new test
- Repeat for another feature



Advantages

- Code coverage: Every piece of code you write will have at least one test that is covering it
- Regression testing: The test suite is developed regressively as you develop
- Simplified debugging: When a test fails, it's obvious where the problem has been introduced
- Alternate documentation: The tests themselves act as a form of documentation that describe what the code should be doing