# Application Architecture

Hönnun og smíði hugbúnaðar

Haust 2022

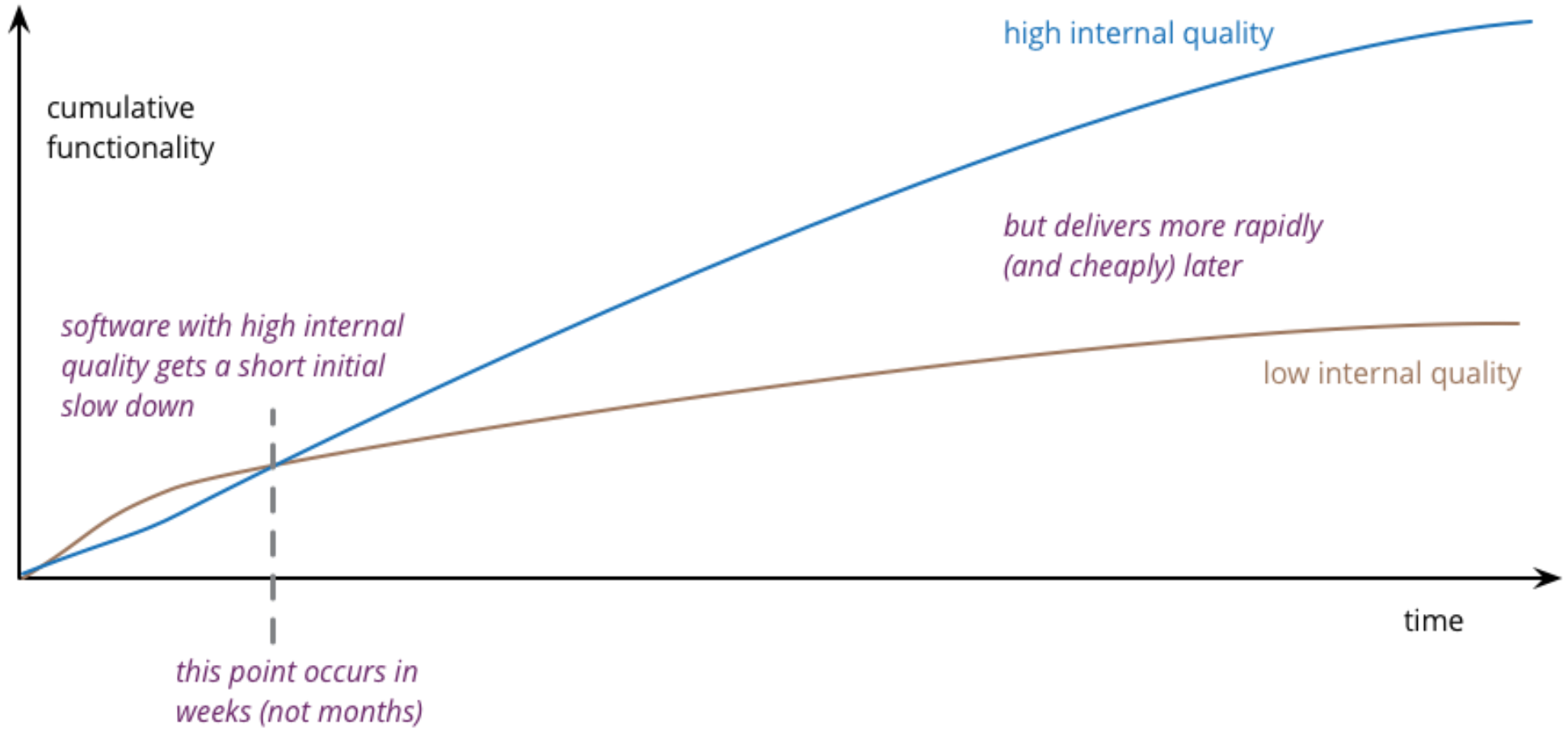
# Hvað er Application Architecture?


- Ekki formlegt heiti
  - EF þú google-ar þá færðu margar skilgreiningar
- Mín skilgreining
  - Hæsti architecture / strúktúr innan ákveðis Independently deployable *application* *(t.d. eitt microservice eða monolith)*
  - Higher level en hönnunarmynstur
  - Rökfastur aðskilnaður á kóða (yfirleitt eftir domain eða tækni)
  - “The shape of the software system”
- Hvað þetta er ekki
  - Ekki architecture á heildar kerfinu
    - Microservice Architecture
    - Monolith
    - Event Driven Architecture
    - Etc.
  - Ekki beint architecture á milli klasa
    - T.d. OOP hönnunarmynstur

# Kostir

- Consistency
  - Viljum vera samkvæm okkur sjálfum og öðrum
- Breytingar eru augljósar
  - Viljum ekki hugsa hvar kóði á að vera, ætti að vera augljóst
  - Myndar uppskrift að lausn
- Flæði forrits augljósara
  - Gefur þér yfirlitsmynd á flæðið og virknina
  - Auðvelt að rekja sig í gegnum kerfið
- Auðvelt að læra á application-ið
  - Ekkert mental mapping ef þú þekkir architecture-inn
- Maintainability
  - Breytingar auðveldari, hættu minni og augljósari

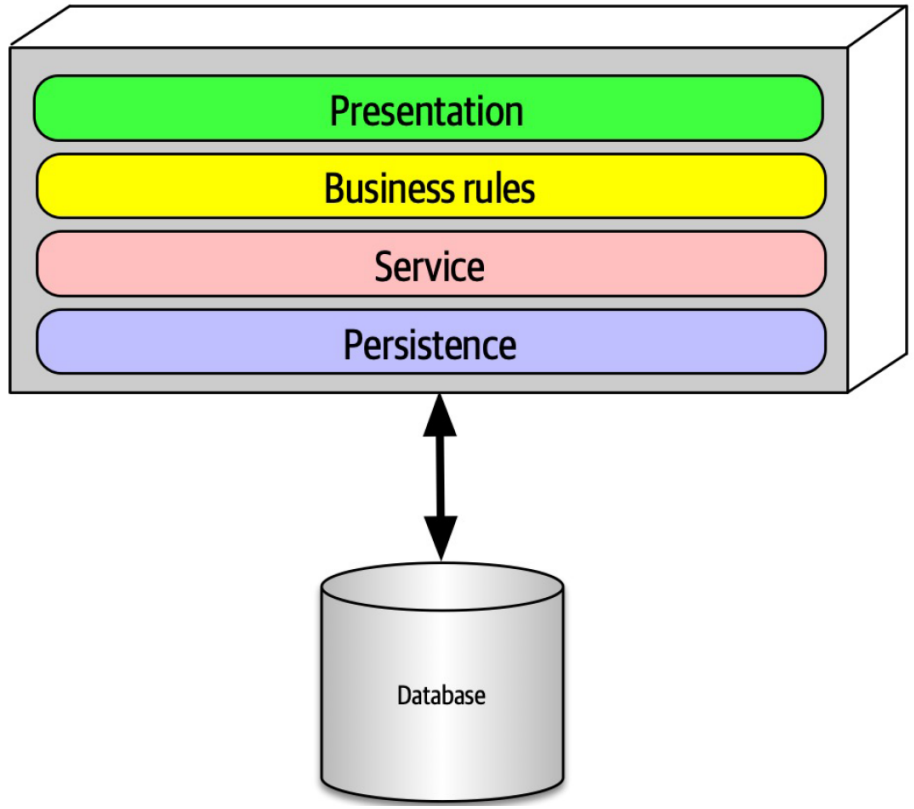
# Kostir frh.


high internal quality
cumulative
functionality
but delivers more rapidly
(and cheaply) later
software with high internal
quality gets a short initial
slow down
low internal quality
time
this point occurs in
weeks (not months)

# Helstu Application Architectures

- N-Tier
- Onion
- Vertical Slicing

# N-Tier Architecture




- Einnig þekkt sem Layered Architecture
- Skipting eftir tæknilegum eiginleikum
  - Horizontal slicing
- Fjöldi laga getur verið hver sem er
  - 3-Tier algengast
    - Presentation / UI Layer
    - Business / Domain Layer
    - Persistence / Repository Layer
- Hvert layer getur bara talað við layer-ið fyrir neðan sig
  - Stuðlar að loose coupling
  - T.d. presentation á ekki að tala við Persistence

# N-Tier Sinkhole Anti-Pattern

- Sinkhole skilgreining
  - Þegar request rennur í gegnum layer-in án þess að layer-in bæta við logic eða virkni
  - T.d. Öll layer-in eru bara þunnir wrapper-ar fyrir gagnagrunninn

- 80-20 reglan
  - Sinkhole mun alltaf gerast að einhverju leyti í layered architecture
  - 80-20 reglan segir að það er ok ef 20% af requestum er sinkholes
  - 80-20 bara viðmið en ekki regla

# N-Tier Kostir og Gallar

- Kostir
  - Einfalt
  - Þekkt
  - Decouples tæknilega þætti
  - Varðveitir SRP og Encapsulate what varies **hvað varðar tækni**
  - Hægt að fylgja Conway's Law

- Gallar
  - Domain logic dreifð
    - **Domain logic á til með að breytast saman**
    - Bort á SRP og Encapsulate what varies **hvað varðar business logic**
  - Brot á DIP
  - Skalast illa

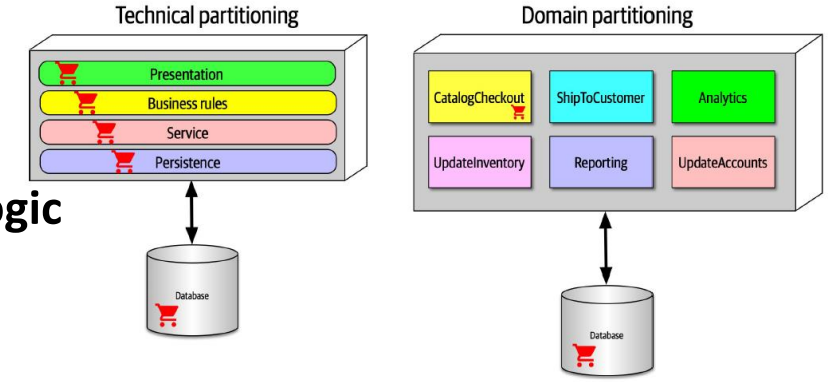



*Figure 8-5. Where domains/workflows appear in technical- and domain-partitioned architectures*

# Conway‘s Law

**CONWAY’S LAW**

Back in the late 1960s, Melvin Conway made an observation that has become known as *Conway’s law*:

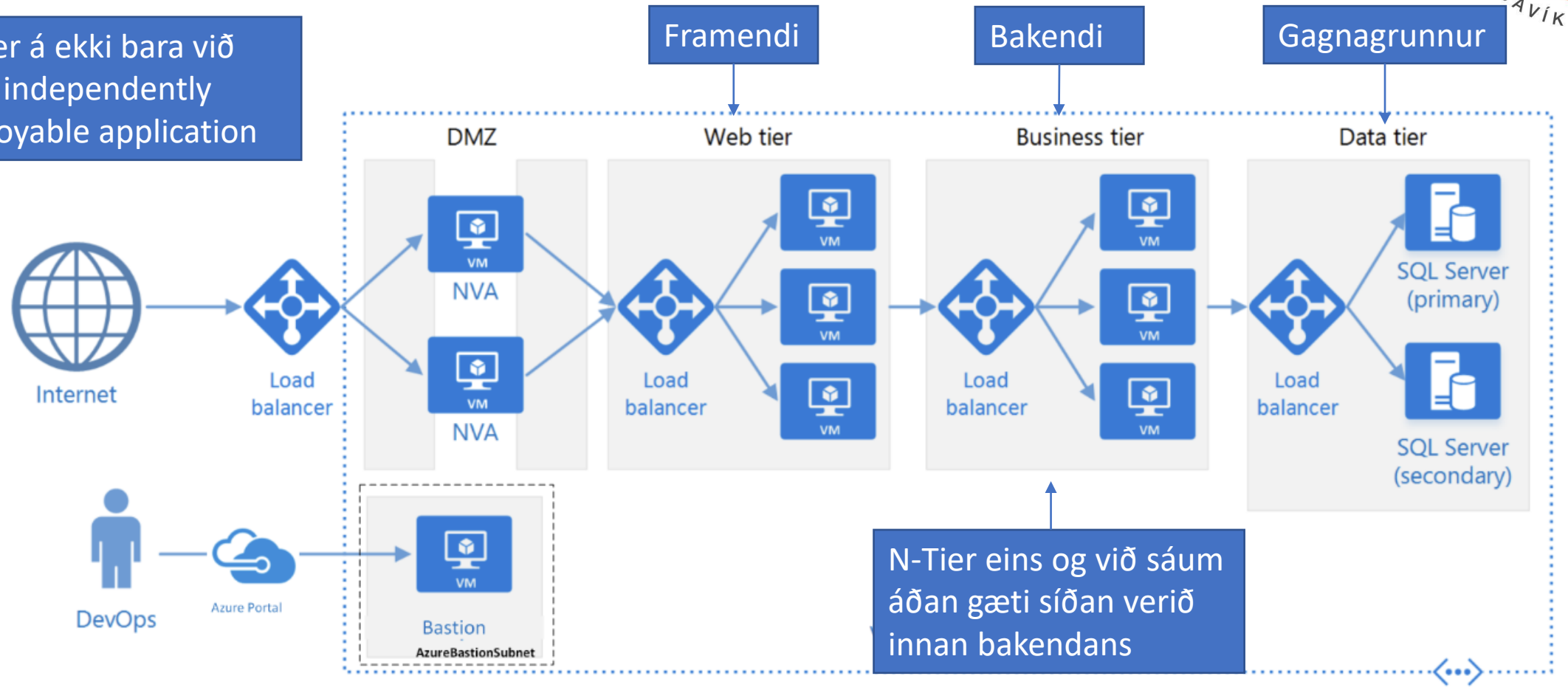
> *Organizations which design systems … are constrained to produce designs which are copies of the communication structures of these organizations.*

Paraphrased, this law suggests that when a group of people designs some technical artifact, the communication structures between the people end up replicated in the design. People at all levels of organizations see this law in action, and they sometimes make decisions based on it. For example, it is common for organizations to partition workers based on technical capabilities, which makes sense from a pure organizational sense but hampers collaboration because of artificial separation of common concerns.

A related observation coined by Jonny Leroy of ThoughtWorks is the Inverse Conway Maneuver, which suggests evolving team and organizational structure together to promote the desired architecture.

# N-Tier Distributed

N-Tier á ekki bara við um í independently deployable application



# N-Tier Pipeline

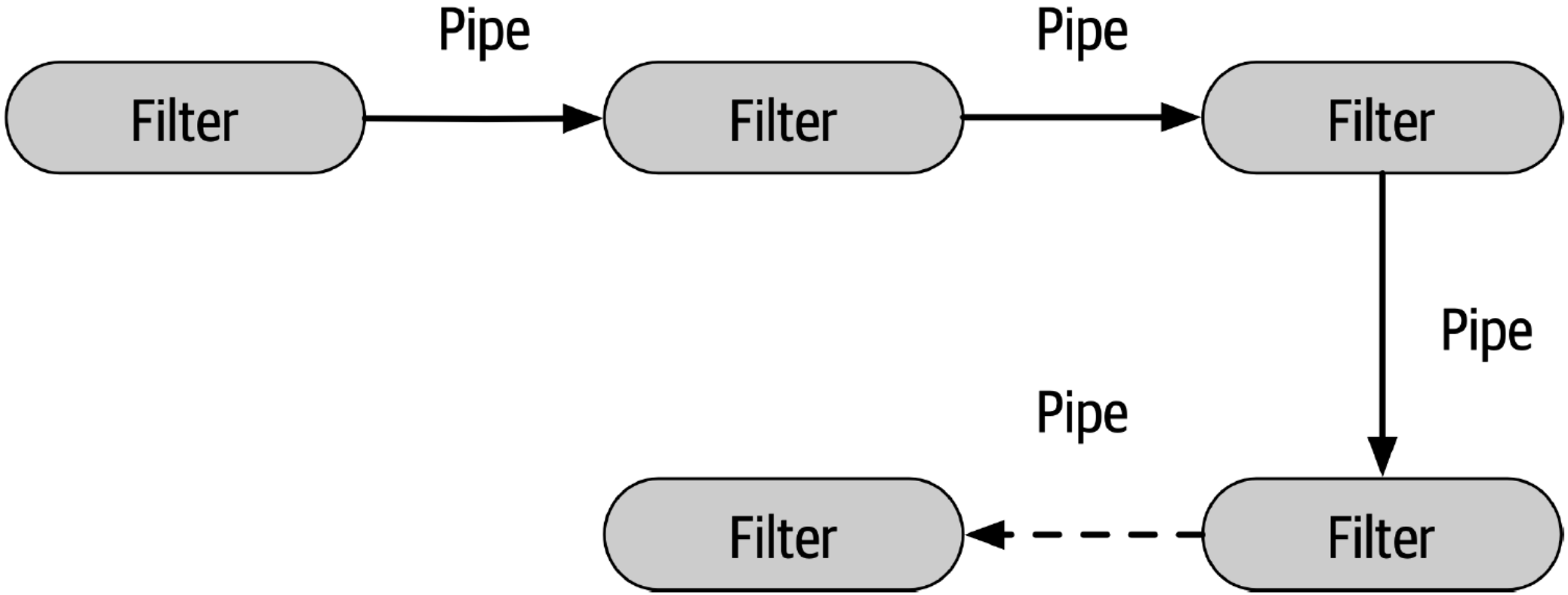
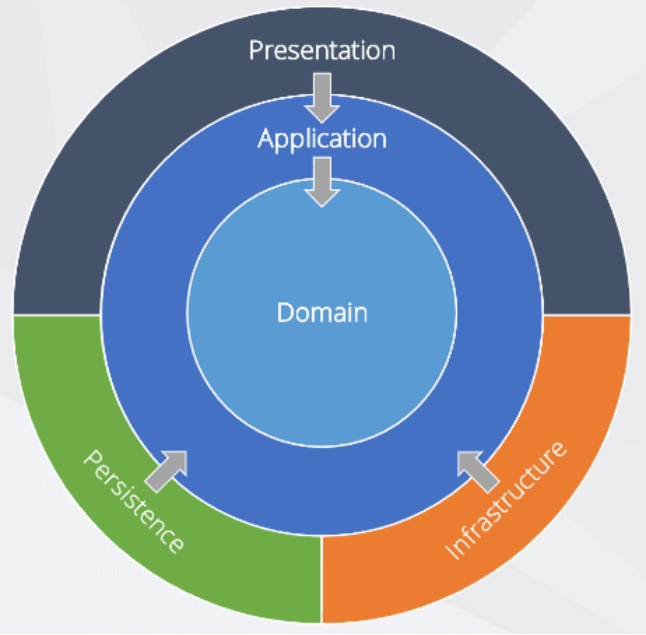



*Figure 11-1. Basic topology for pipeline architecture*

# Onion Architecture




- Einnig þekkt sem
  - Hexagonal Architecture
  - Clean Architecture

- Skipting eftir tæknilegum eiginleikum
  - Horizontal slicing
  - Layered að því leiti
  - Hefur aðrar hugmyndir / constraints um hvernig layer tala saman

- Hugsað sem hringur
  - Innri layer vita ekki um ytri layer
  - Ytri layer depend-a á innri en ekki öfugt *(Direction of Coupling* er í átt að kjarna*)*
  - Skiptist aðallega í
    - **Core** -> (Domain, Application)
    - **Infrastructure** -> (Persistence, External Gateways etc.)
    - **Presentation** -> (Controllers, Endpoints, UI etc.)
  - Innri layer skilgreina interface fyrir ytri til að útfara -> **DIP**

# Onion Architecture Core


- Core
  - Inniheldur Business/Domain model-ið
  - Business/domain logic-ina
  - Inniheldur entitites, application exceptions, domain events, lower level / external interfaces etc.
  - Hefur engin dependency á önnur layer

- Domain Layer
  - Domain Entities (T.d. Order, Buyer, Merchant etc.)
  - Aggregates
  - Domain Exceptions
  - Domain Events

- Application Layer
  - Business Logic
  - Application Exceptions
  - External Interfaces
  - Hefur dependency á domain layer-ið

- Domain og Application layer **oft sameinuð í eitt application core layer**

# Onion Architecture Presentation

- Inngangurinn í kerfið
  - Getur verið API -> Endpoints, Controllers etc.
  - Getur verið UI kóði -> HTML, Javascript, CSS etc.

- Hefur Dependency á Core

- Notar interface skilgreind í Core
  - Notar útfærðan kóða í core eða infrastructure layers með því að nota interface fyrir þau skilgreind í Core.

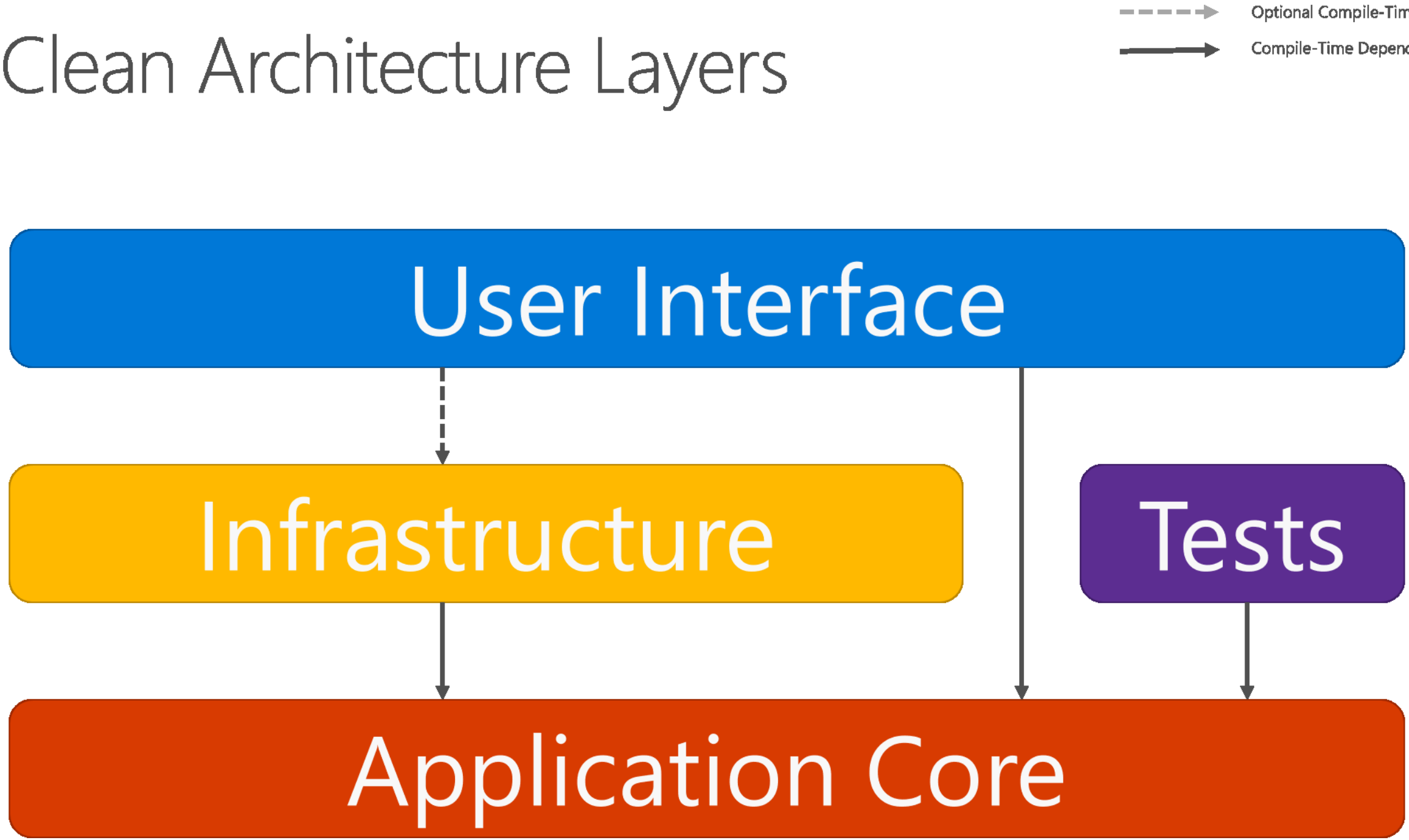
# Onion Architecture Infrastructure


- Inniheldur samþættingu við external hluti
  - T.d. Gagnagrunnur, External API, Twilio, Sendgrid, Elasticsearch etc.
- Hefur Dependency á Core
- Útfærir interface skilgreind í Core

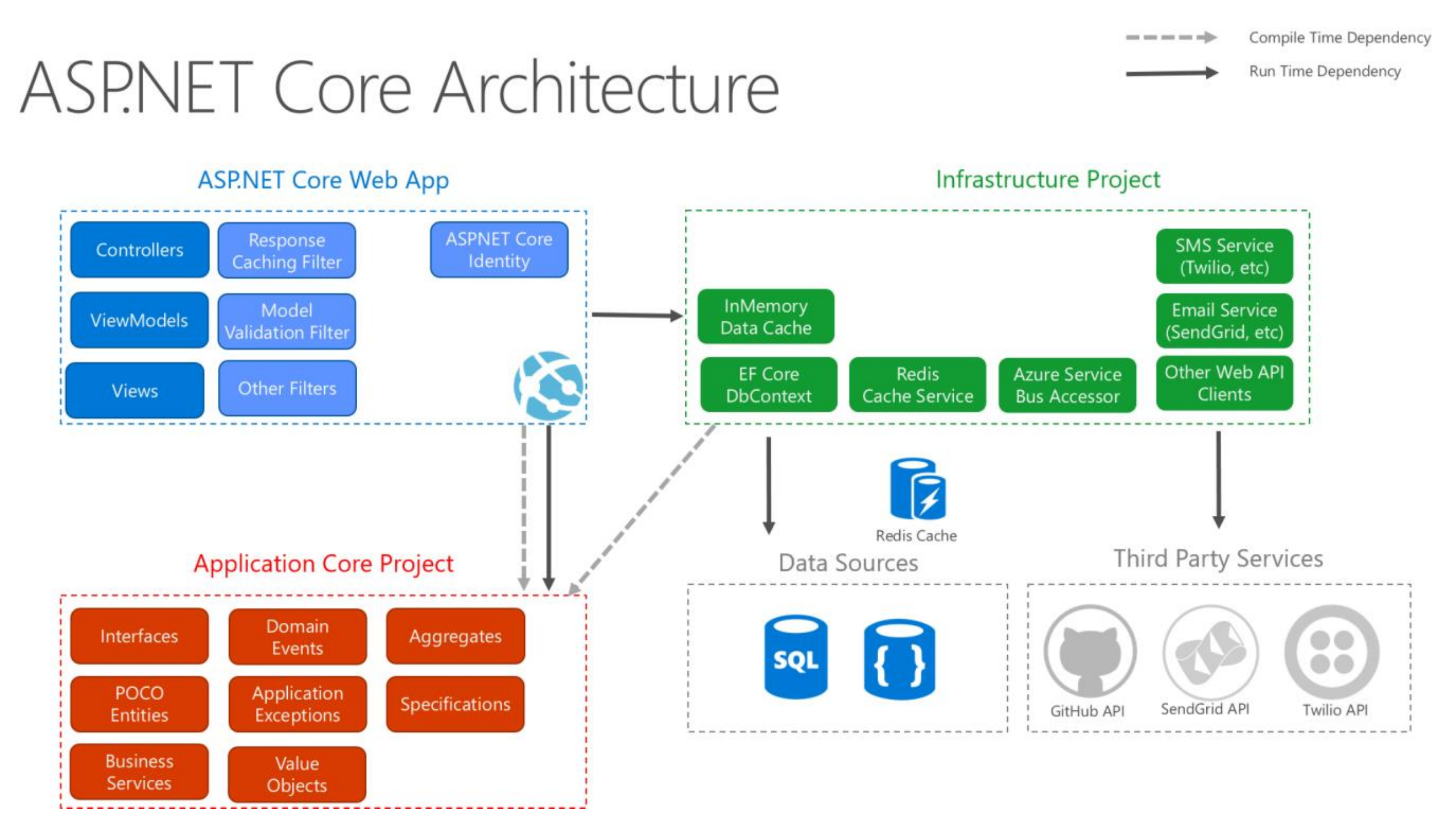
# Onion Architecture DIP

- Onion stuðlar að Dependency Inversion Principle
  - *„high-level modules should not depend on low-level modules. Both should depend on abstractions“*
  - *„Abstractions should not depend on details. Details should depend on abstractions“*
- Business logic á að knýja breytingar á lower level components
- Core er high level layer sem sér um business logic
- Core á að vera decoupled frá *detail* breytingum í infrastructure layer
- Core og interface
  - Core skilgreinir þau interface sem infrastructure layer-ið á að útfæra
  - Core skilgreinir þessi interface **út frá sínum business þörfum**

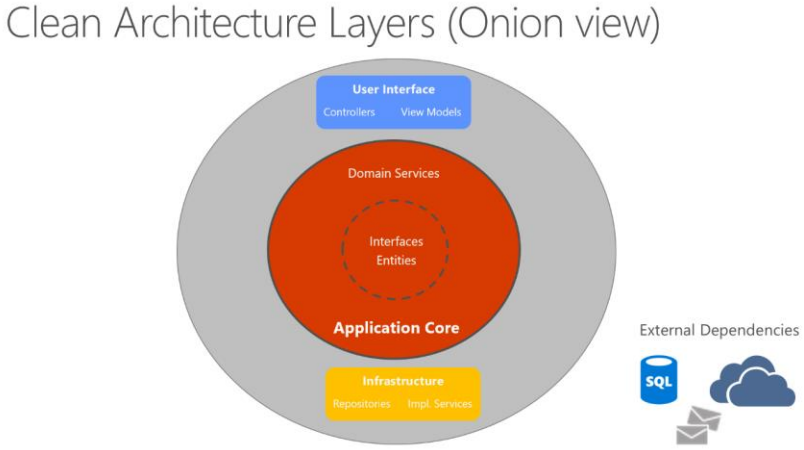
# Clean Architecture Layers

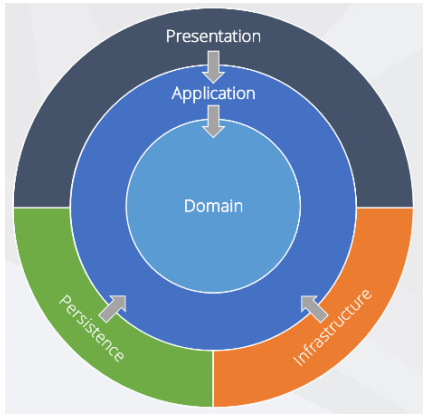


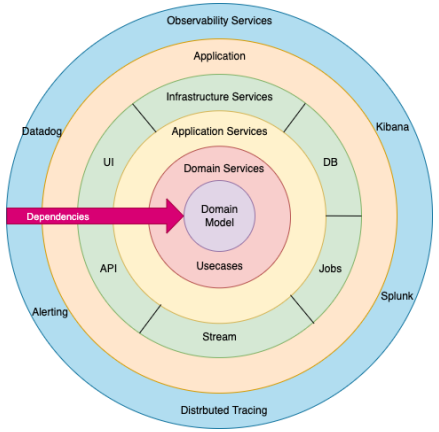
# ASP.NET Core Architecture

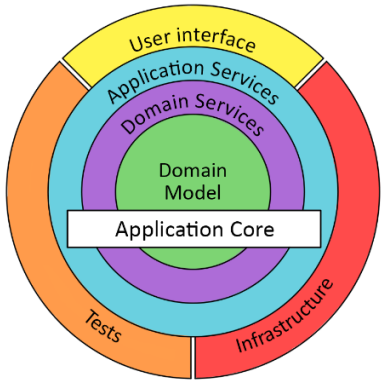


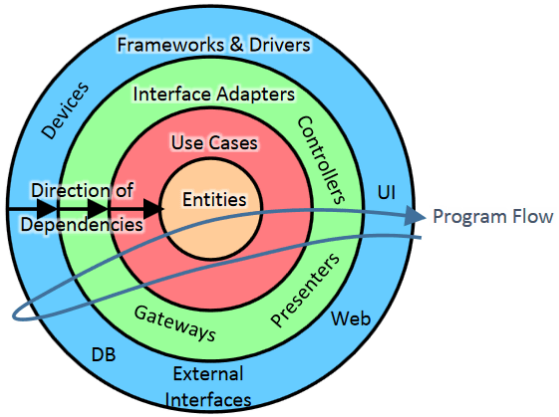
# Mörg Diagram en sama hugmynd

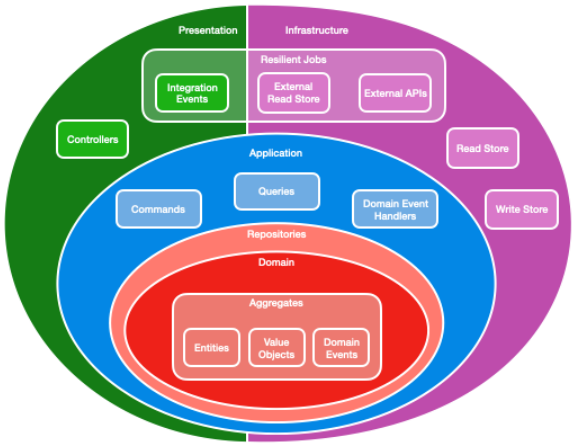


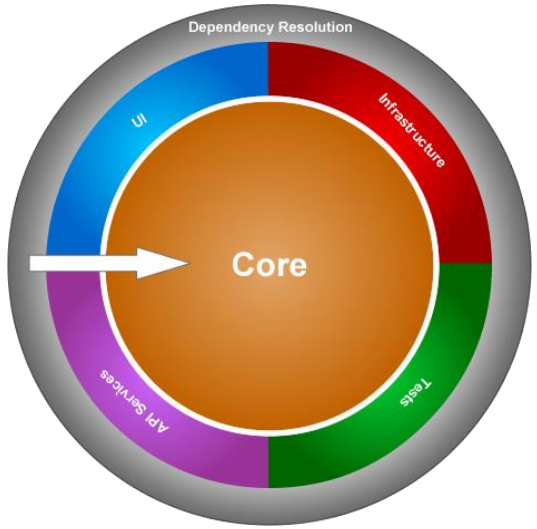



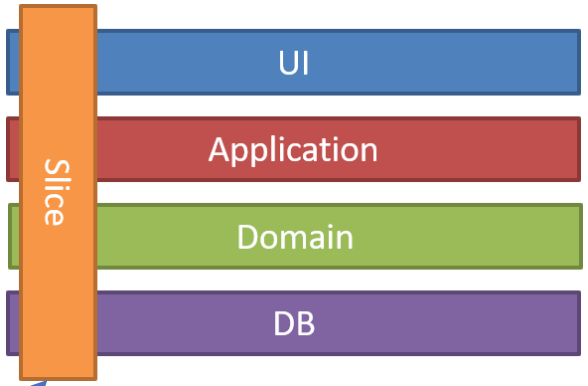



# Onion Architecture Kostir og Gallar

- Kostir
  - Stuðlar að DIP
  - Business logic sem knýir breytingar
  - Breytingar verða náttúrlegri þegar þær eru gerðar út frá business logic
  - Sömu kostir á Layered Architecture
  - Skalast vel

- Gallar
  - Mikið boilerplate
  - Flóknara
  - Overkill fyrir lítil verkefni
  - Abstraction-in í Core oft röng / þurfa að breytast hvort sem er
  - Domain logic dreifð

# Vertical Slicing




- Einnig þekkt sem Modular Architecture
- Skiptum kóða eftir domain eiginleikum
  - *Vertical slices*
  - Allur kóði fyrir sama feature / subdomain hópaður saman
  - Presentation, Business og Persistence logic undir sama *slice*
- Breytingar eiga til með að snerta á mörgum *layers* fyrir eitt domain
- **Decoupling á milli domain-a betri** decoupling en á milli layers
- *Minimize coupling between slices, and maximize coupling in a slice*
- Höfum minni áhyggjur af coupling milli laga og því **minin þörf fyrir flókin boilerplate abstractions**

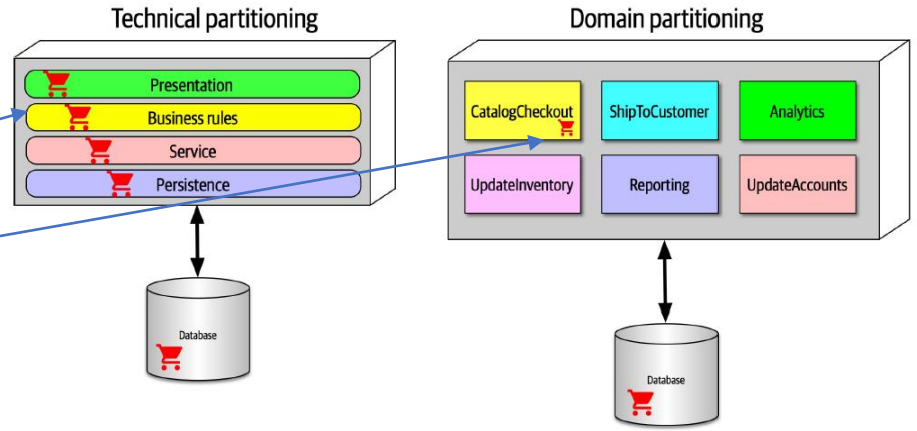



*Figure 8-5. Where domains/workflows appear in technical- and domain-partitioned architectures*

# Vertical Slicing Vandamál

- Hvað með cross-cutting concerns á milli slice-a?
  - T.d. Logging, Error handling, DI, configuration, pressentation config, database config etc.
  - **Lausn ->** Sameiginlegt *common* slice sem hin slice-in nota.

- Hvað með domain logic sem tvö slice þurfa?
  - **Lausn 1 ->** Annað slice-ið hefur þessa logic og expose-ar virkninni í gegnum interface fyrir hitt slice-ið til að nota
  - **Lausn 2 ->** Brjóta sameiginlegu virknina í annað slice og expose-a með interface-um

# Vertical Slicing Kostir og Gallar

- Kostir
  - Einfalt
  - Breytingar eiga til með að vera hjúpaðar innnan ákveðins domains
  - Strúktúr augljós
  - Oft ekki þörf á flóknum abstractions
  - Auðveldar að brjóta upp í microservices
  - Hátt cohesion innan hvert slice
- Gallar
  - Endurtekt
  - Þurfa að deila cross-cutting concerns kóða
  - Samskipti / deildur kóði á milli domain-a erfiðara

# Hybrid

- Hver er architecture-inn innan core í Onion?
- Hver er architecture-inn innan hvers slice í Vertical Slicing?

- Getum blandað saman architecter-um

- Slice í vertical slicing gæti haft innri architecture sem onion eða N-Tiered

- Layers í onion gætu haft innri architecture sem vertical slicing eða N-Tiered

# Annað

- Domain Driven Design
  - https://martinfowler.com/bliki/DomainDrivenDesign.html
  - https://awesome-architecture.com/domain-driven-design/domain-driven-design/
- Repository Pattern
  - https://docs.microsoft.com/en-us/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/infrastructure-persistence-layer-design
  - https://www.youtube.com/watch?v=01lygxvbao4&ab_channel=CodeOpinion
- REPR
  - https://fast-endpoints.com/
- CQRS
  - https://docs.microsoft.com/en-us/azure/architecture/patterns/cqrs
  - https://awesome-architecture.com/cqrs/
- State Management
  - Algengast í framenda
  - BLoC, Redux, Store, …
  - Margt í boði farandi eftir hvað þú ert að gera
  - https://docs.flutter.dev/development/data-and-backend/state-mgmt/options
- MVC, MVP, MVVM
  - https://levelup.gitconnected.com/mvc-vs-mvp-vs-mvvm-35e0d4b933b4#:~:text=References%20%E2%80%94%20In%20MVC%2C%20the%20View,entry%20point%20is%20the%20View.
- Ooooooog svo mikið meira