Patterns of Software Architecture

Software Engineering

Alessio Gambi - Saarland University

Based on the work of Cesare Pautasso, Christoph Dorn, and their students

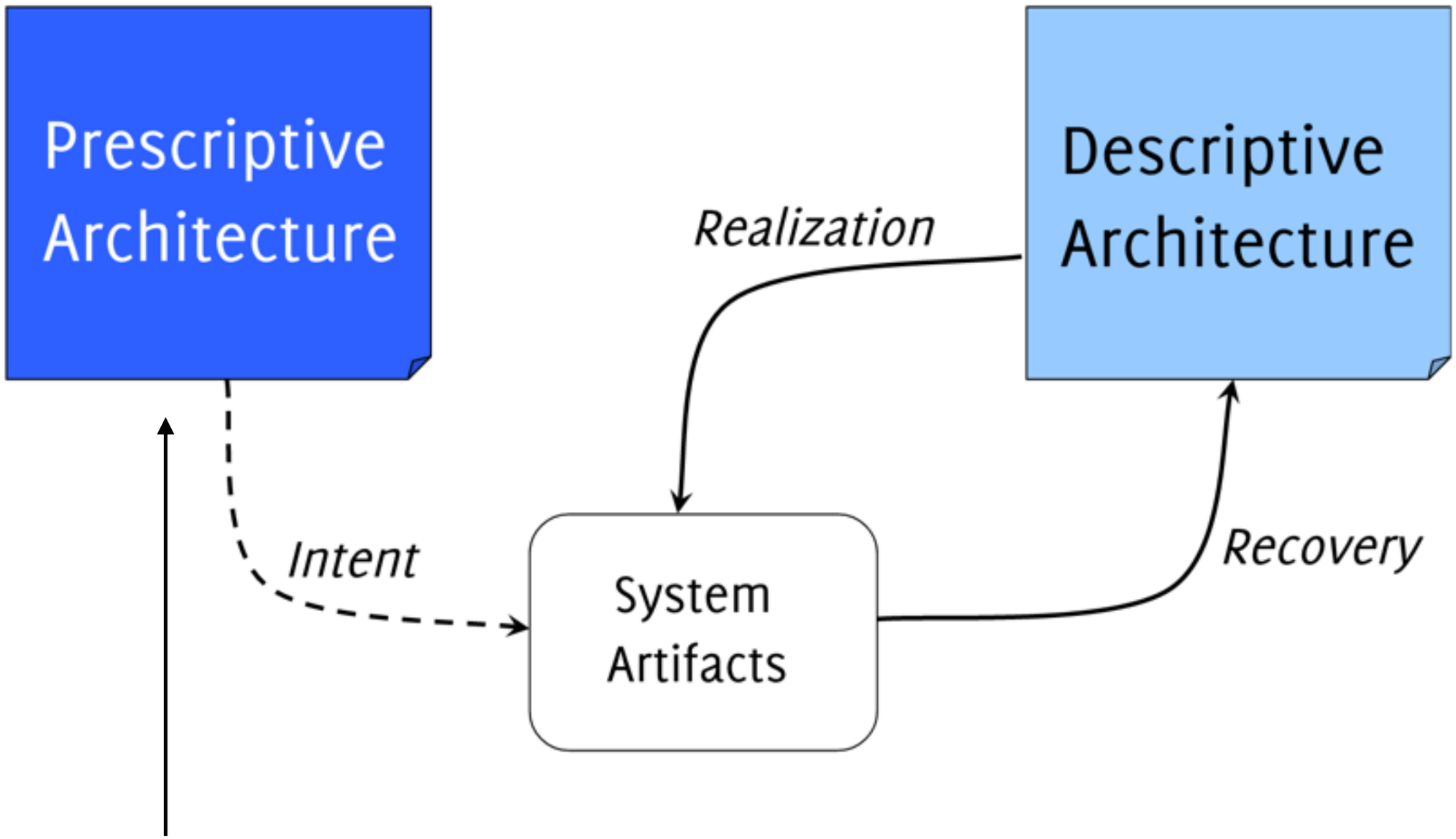
Software Architecture

A software system's architecture is the set of principal design decisions made about the system.

N. Taylor et al.

Abstraction
Communication
Visualization and Representation
Quality Attributes

Every system has a software architecture



What designers want

Modeling

- Problem

  Domain model

- Environment

  System Context

  Stakeholders

- System-to-be

  Boundary/Internal Model

  Quality attributes

  Development

- Components

  Computation

  State

- Connectors

  Interaction

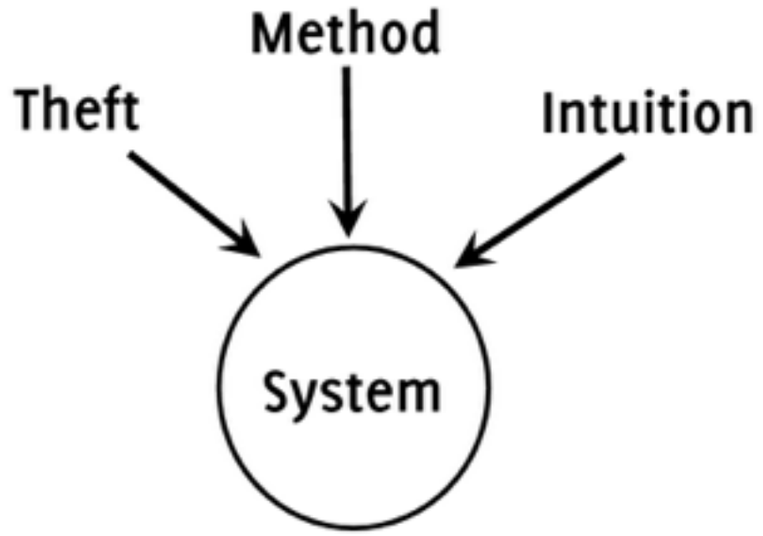
  C. C. C. F.

- Views & Viewpoints

  Kruchten 4+1

Design

- Architectural Styles
- Architectural Patterns
- Building Blocks
  - *Software Connectors*



Architectural Styles

Named collections of architectural decisions that are applicable in a development context.

They constrain architectural design decisions, are specific to the system within that context, and elicit beneficial qualities in each resulting system

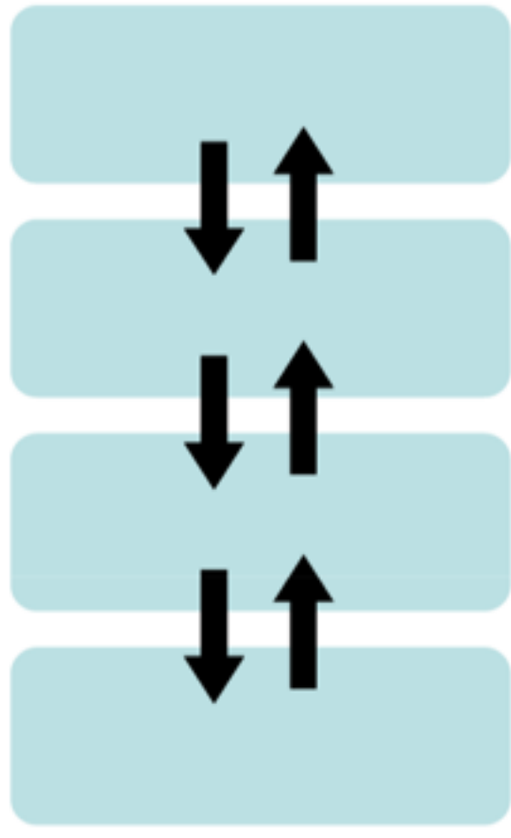
Monolithic

- Lack of structure
- No Constraints
- Poor Maintainability
- Possibly Good Performance

Mainframe COBOL programs · powerpoint · many games

Layered

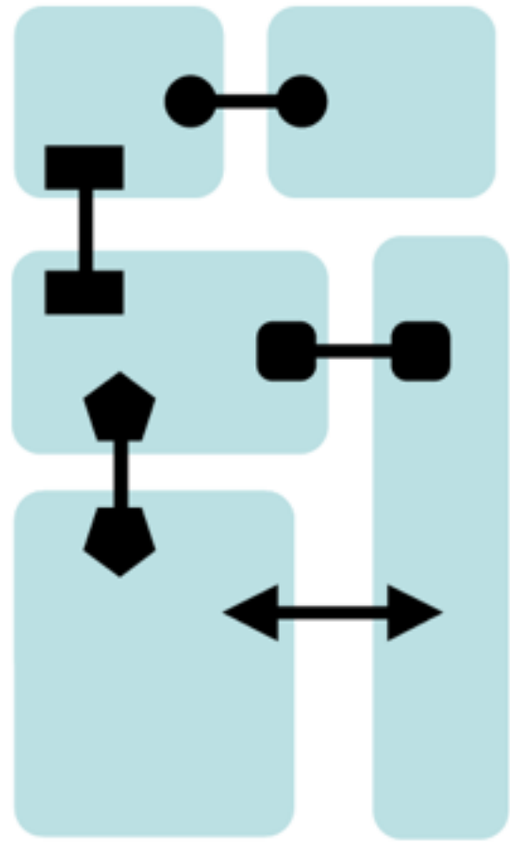
- Communications 1 layer up/down
- Information hiding, no circular deps
- Possibly bad performance
- Good evolvability



Network protocol stacks · Web applications · Virtual Machines

Component Based

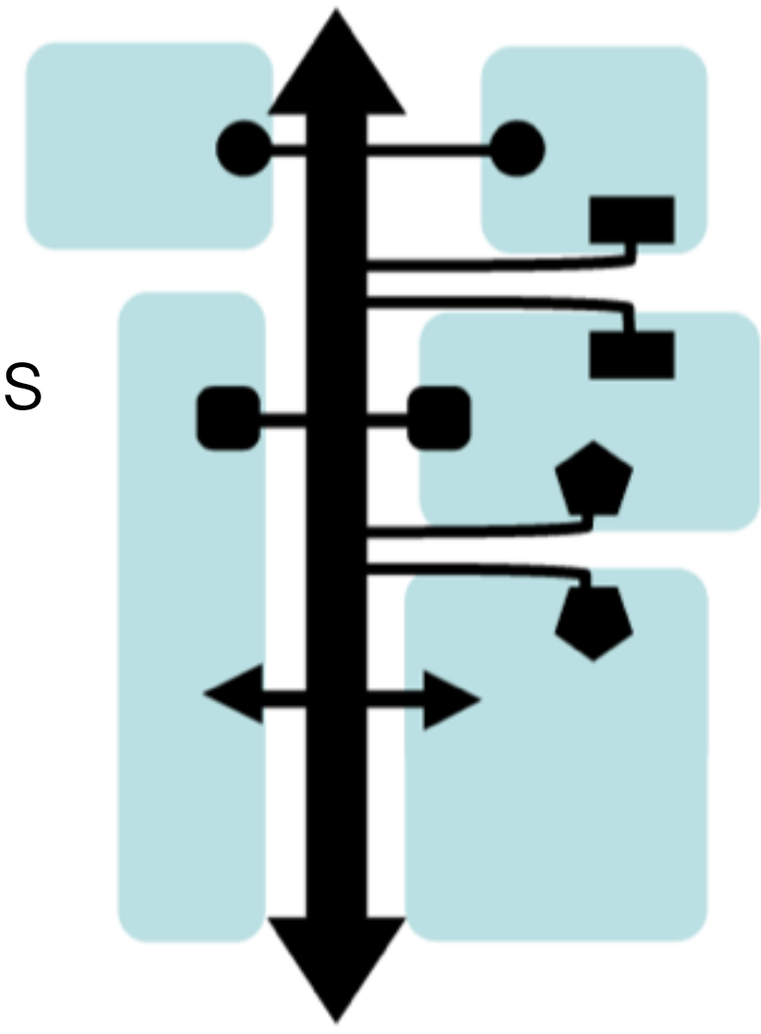
- Encapsulation
- Information hiding
- Components compatibility problem
- Good reuse, independent development



CORBA · Enterprise JavaBean · OSGi

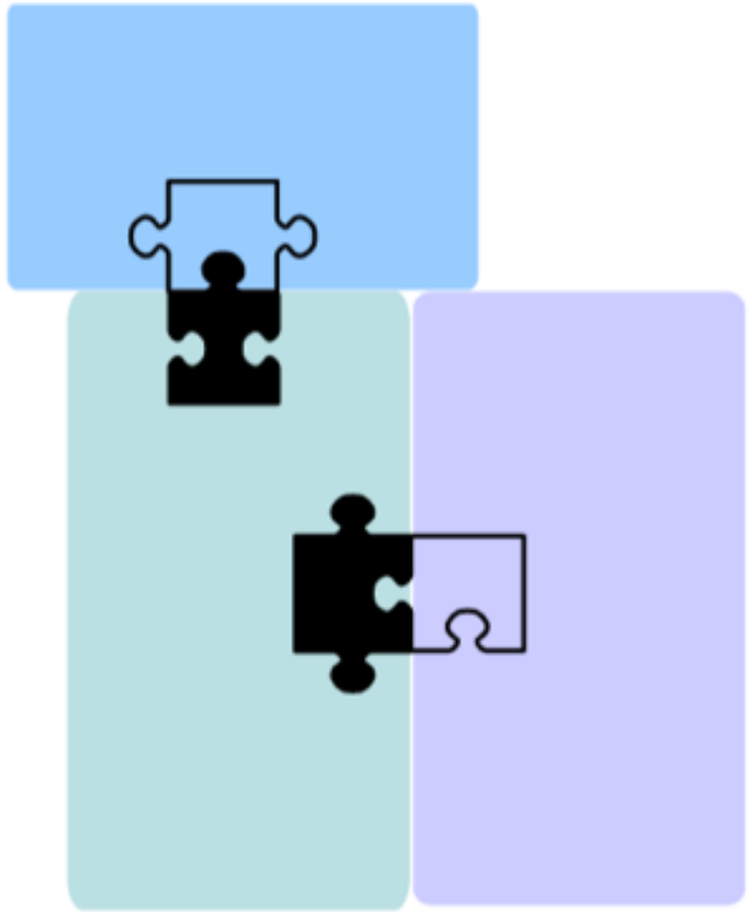
Service Oriented

- Components might be outside control
- Standard connectors, precise interfaces
- Interface compatibility problem
- Loose coupling, reuse



Plugin

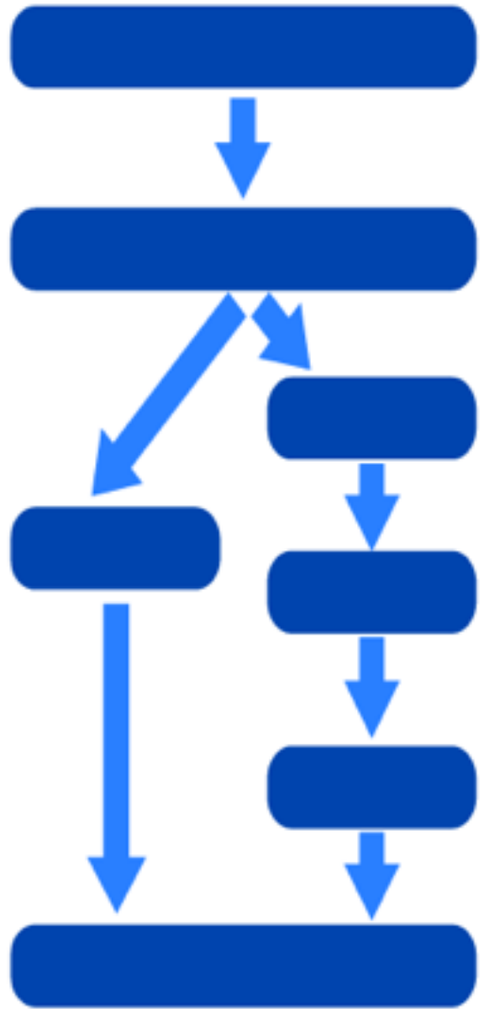
- Explicit extension points
- Static/Dynamic composition
- Low security (3rd party code)
- Extensibility and customizability



Eclipse · Photoshop · Browsers' extensions

Pipe & Filter

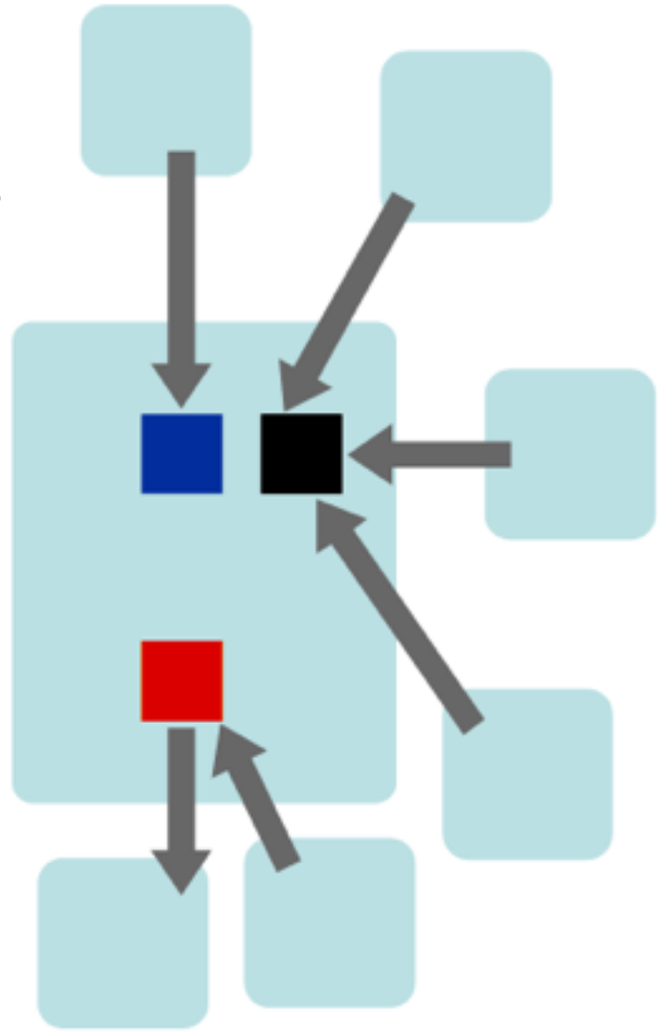
- Clean separation: filter process, pipe transport
- Heterogeneity and distribution
- Only batch processing, serializable data
- Composability, Reuse



UNIX shell · Compiler · Graphics Rendering

Black Board

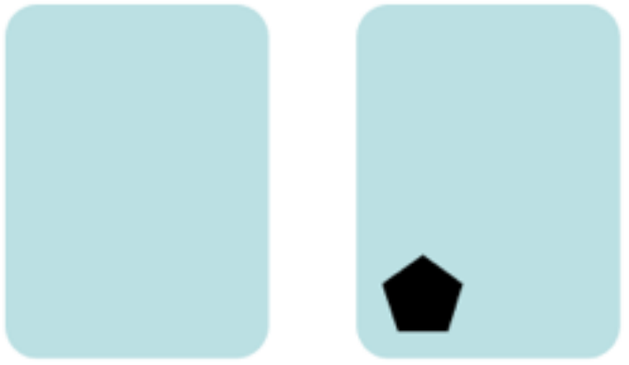
- Collective problem solving via shared data
- Asynchronous components interactions
- Requires common data format
- Loose coupling, implicit data flow

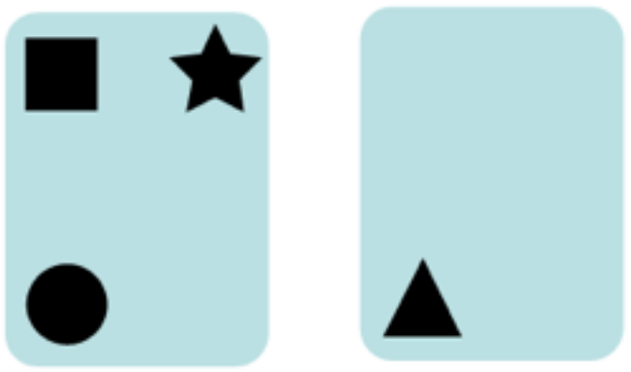


Database · Tuple space · Expert systems (AI)

Event Driven

- Produce/React to events
- Asynchronous signals/messages
- Difficult guarantee performance
- Loose coupling, scalable

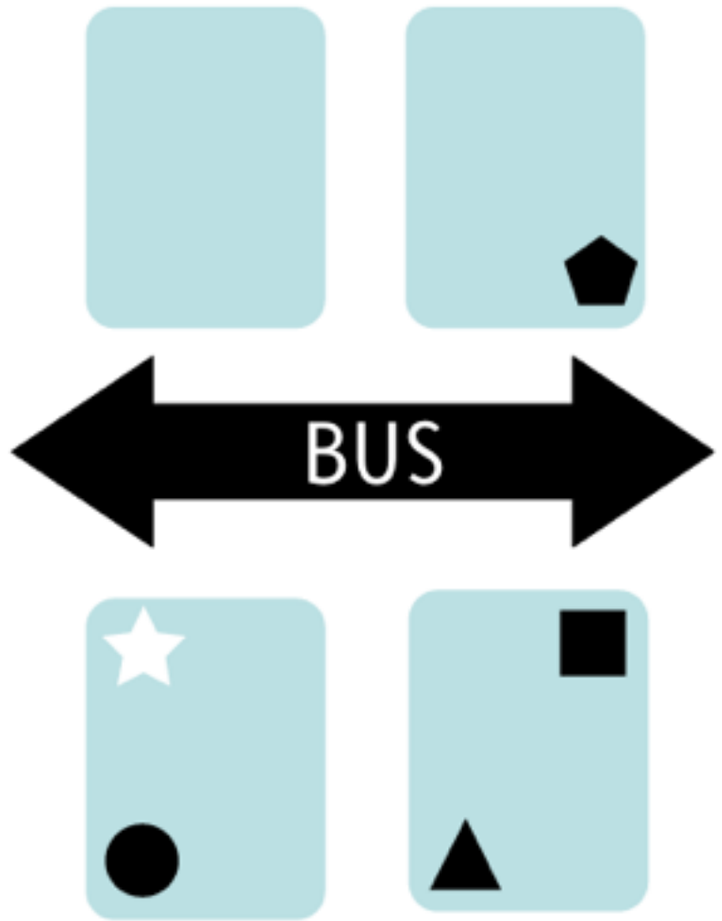




Sensor Monitoring · Complex Event Processing

Publish/Subscribe

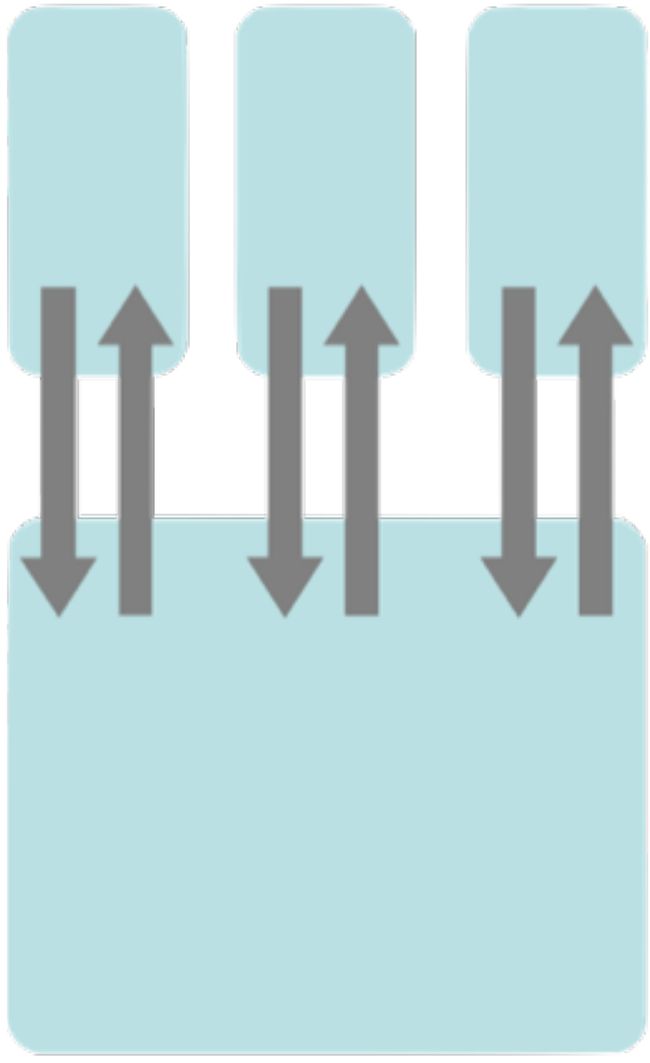
- Event driven + opposite roles
- Subscription to queues or topics
- Limited scalability
- Loose coupling



Twitter · RSS Feeds · Email

Client/Server

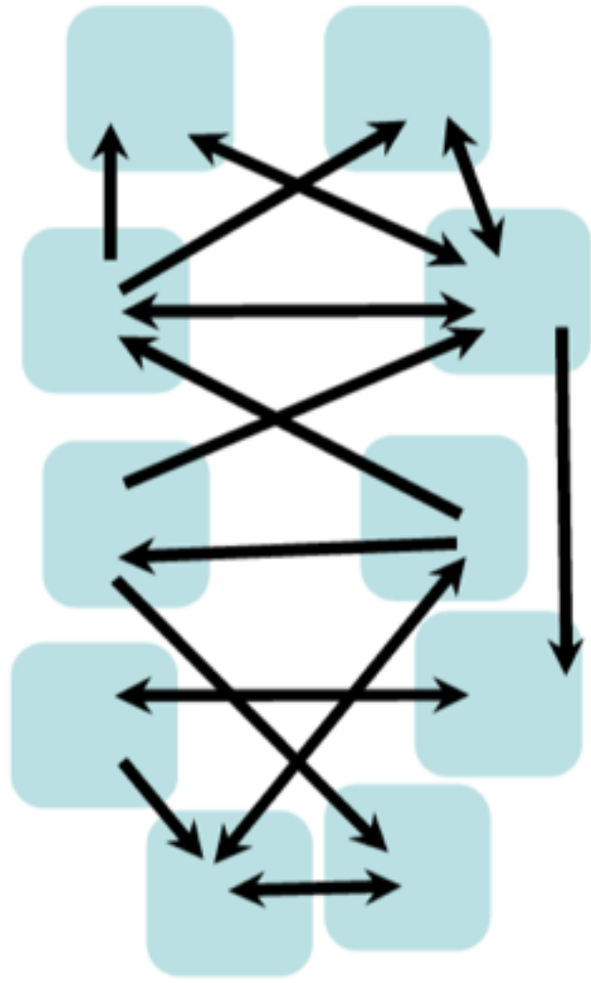
- Many clients, active, close to users
- One server, passive, close to data
- Single point of failure, scalability
- Security, scalability



Web Browser/server · Databases · File Servers · Git/SVN

Peer to Peer

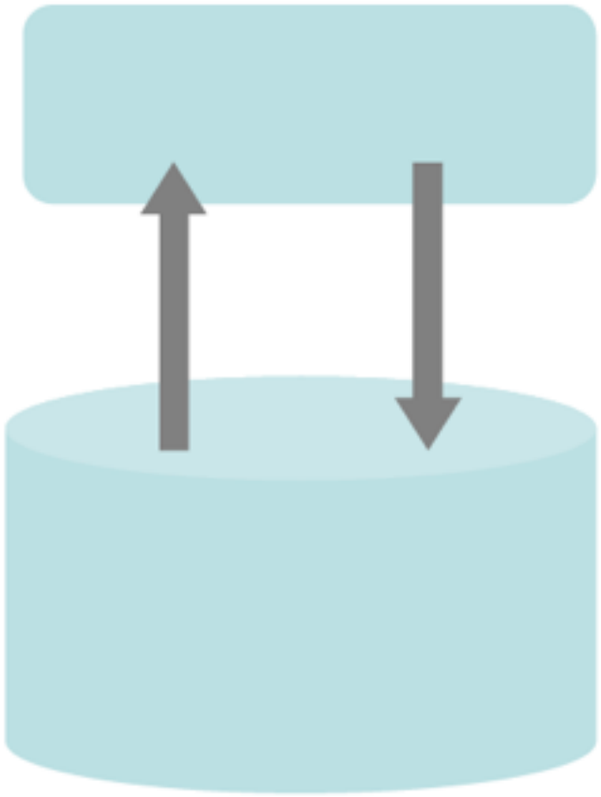
- Both server and client at the same time
- Dynamic join/leave
- Difficult administration, data recovery
- Scalability, dependability/robustness



File Sharing · Skype (mixed style) · Distributed Hash Tables

Data Centric

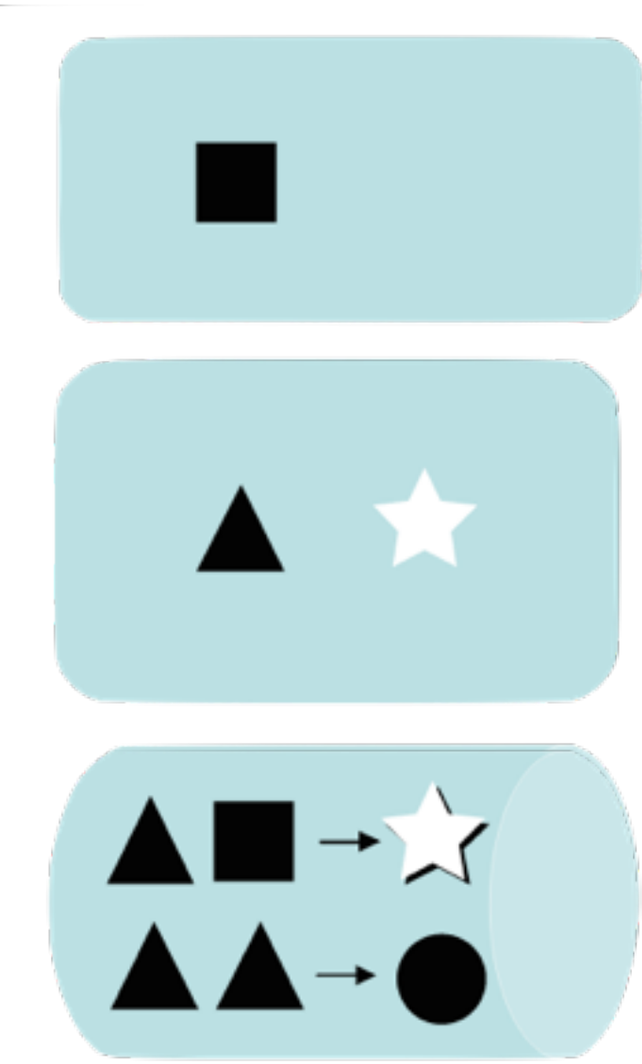
- Persistence layer
- Black board like
- Single point of failure
- (Eventual) Consistency (BASE/ACID)



Relational DB · Key-Value Stores

Rule Based

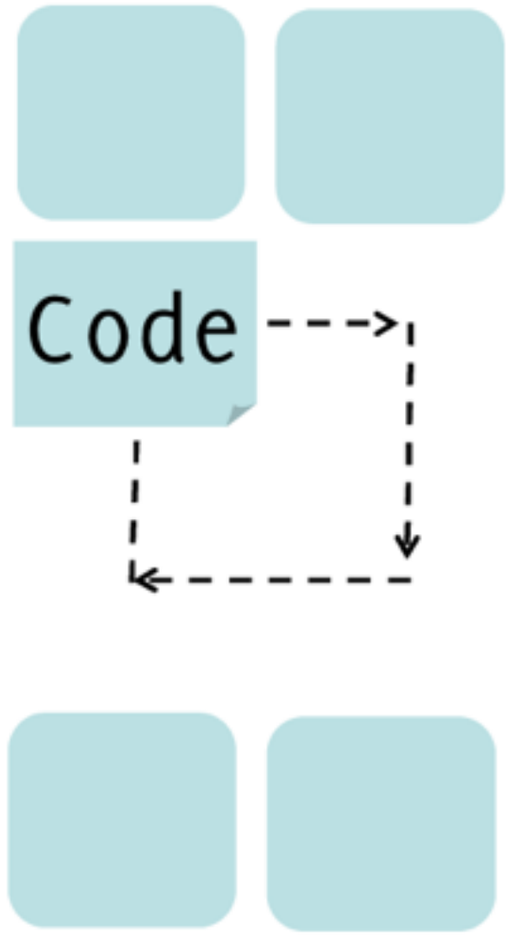
- Rules dynamically triggered
- Layered
- Possibly hard to understand and maintain
- Evolvability



Business Rule Engines · Expert Systems · Prolog

Mobile Code

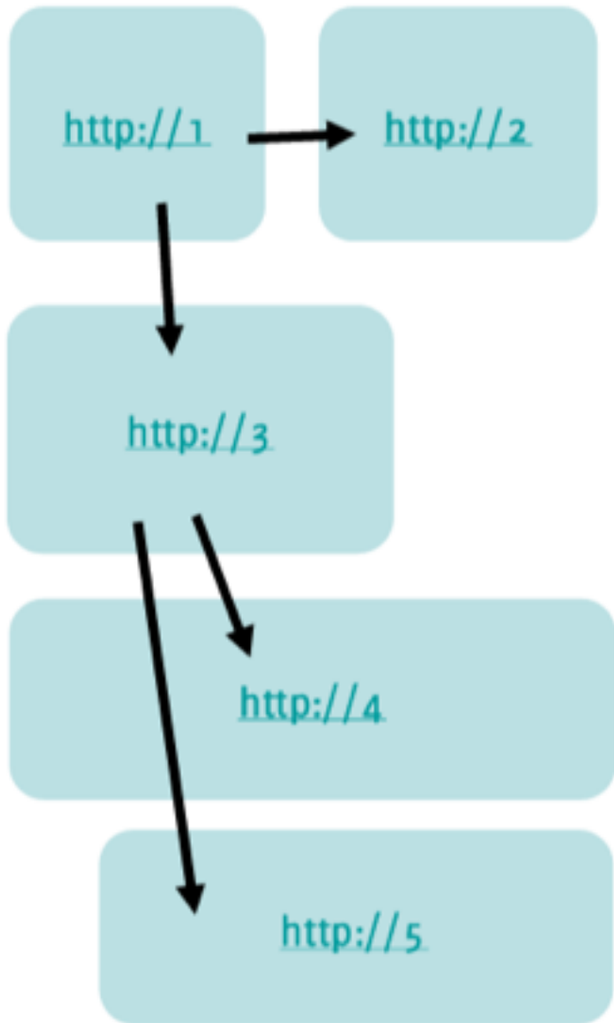
- Code migrates (weak)
- Code+execution state migrate (strong)
- Security
- Fault tolerance, performance



JavaScript · Flash · Java Applets · Mobile Agents · Viruses

REST

- Hybrid style
- Stateless interactions/Stateful resources
- Loose coupling, scalability, interoperability



World Wide Web · RESTFul Web APIs

Architectural Patterns

An architectural pattern is a set of architectural design decisions that are applicable to a recurring design problem, and parameterized to account for different software development contexts in which that problem appears.

Layered - Component - Events - Composition

Layered Patterns

- State-Logic-Display

  separate elements with different rate of change

- Model-View-Controller

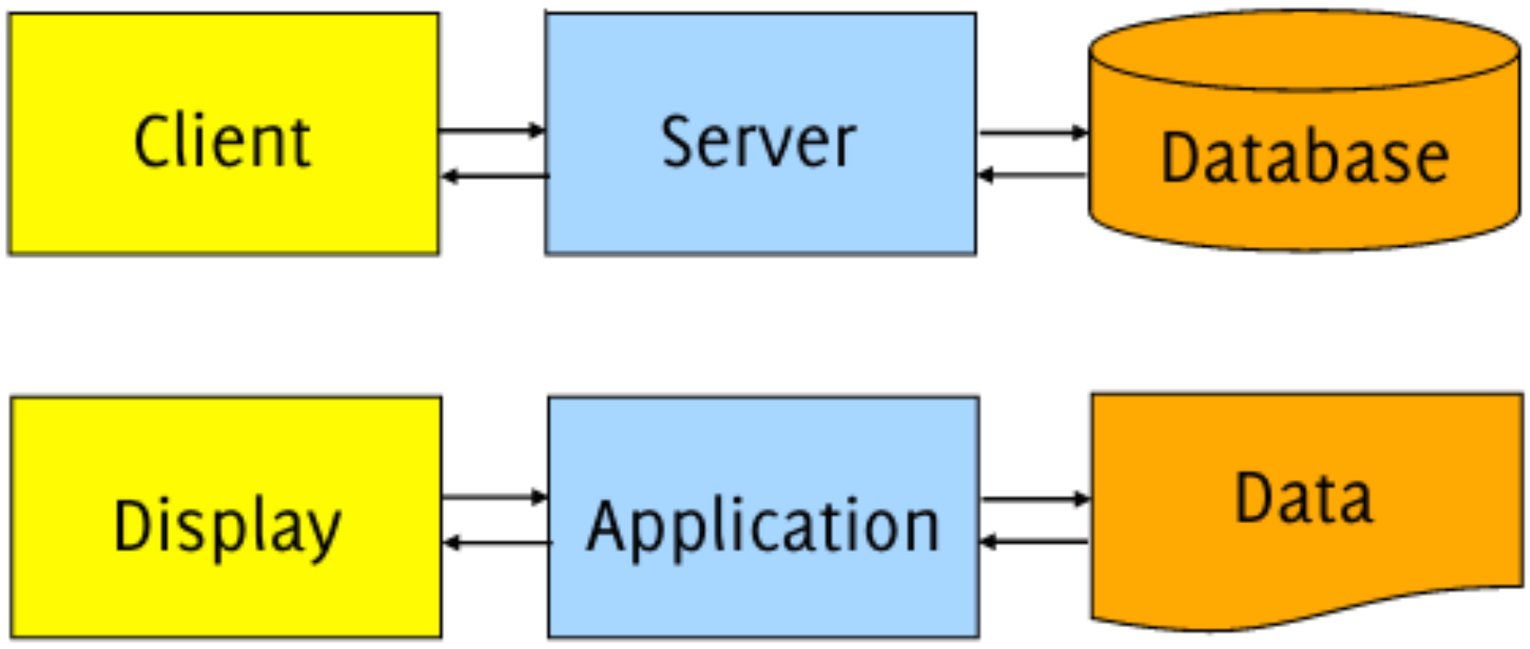
  support many interaction and display modes for the same content

- Presenter-View

  keep a consistent look and feel across a complex UI

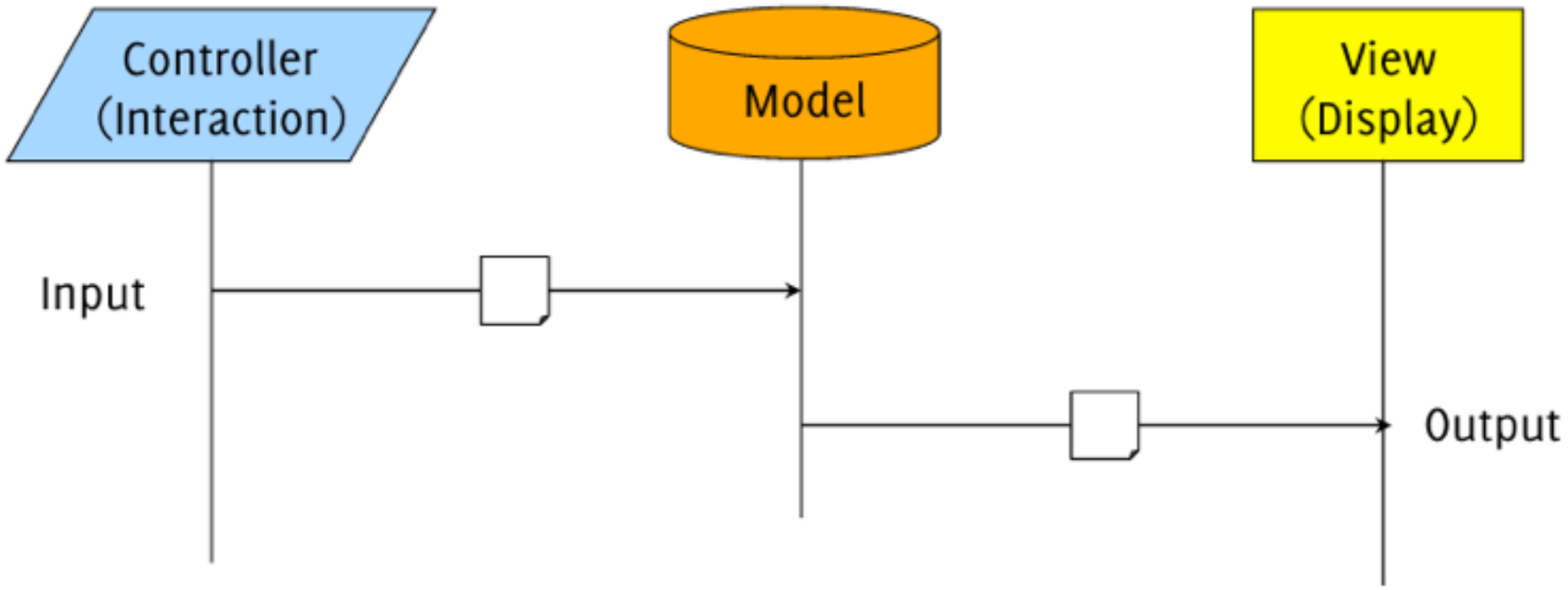
State-Logic-Display

cluster elements that change at the same rate



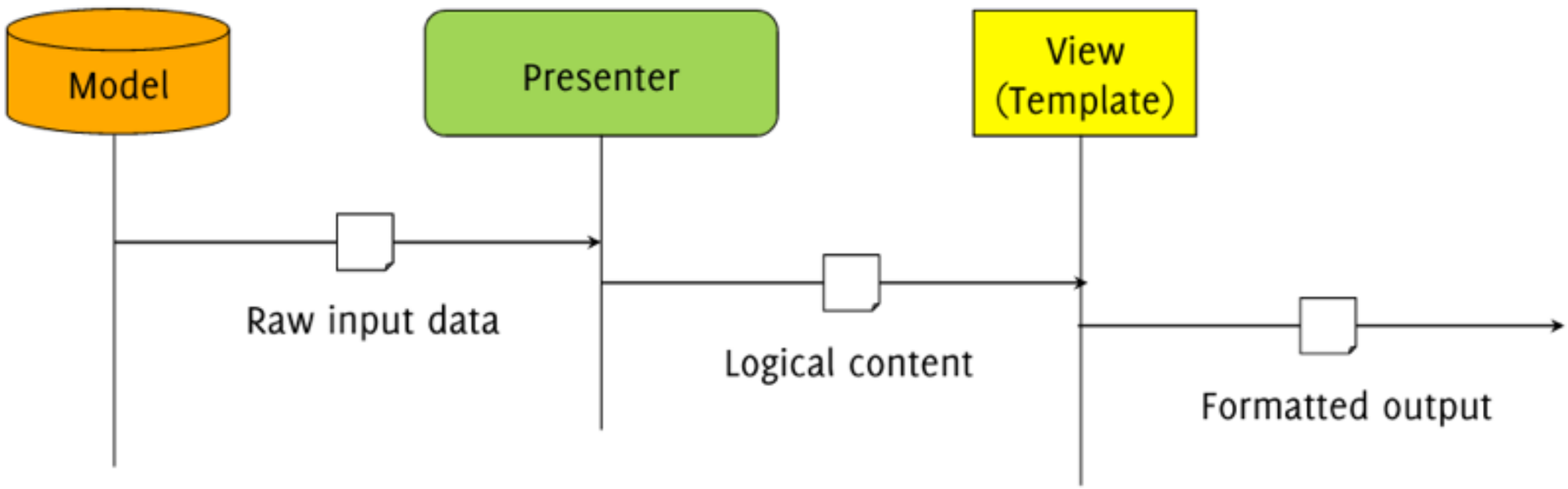
Model-View-Controller

separate content (model) from presentation (output) and interaction (input)



Presenter-View

extract the content from the model to be presented from the rendering into screens/web pages



Component Patterns

- Interoperability

  enable communication between different platforms

- Directory

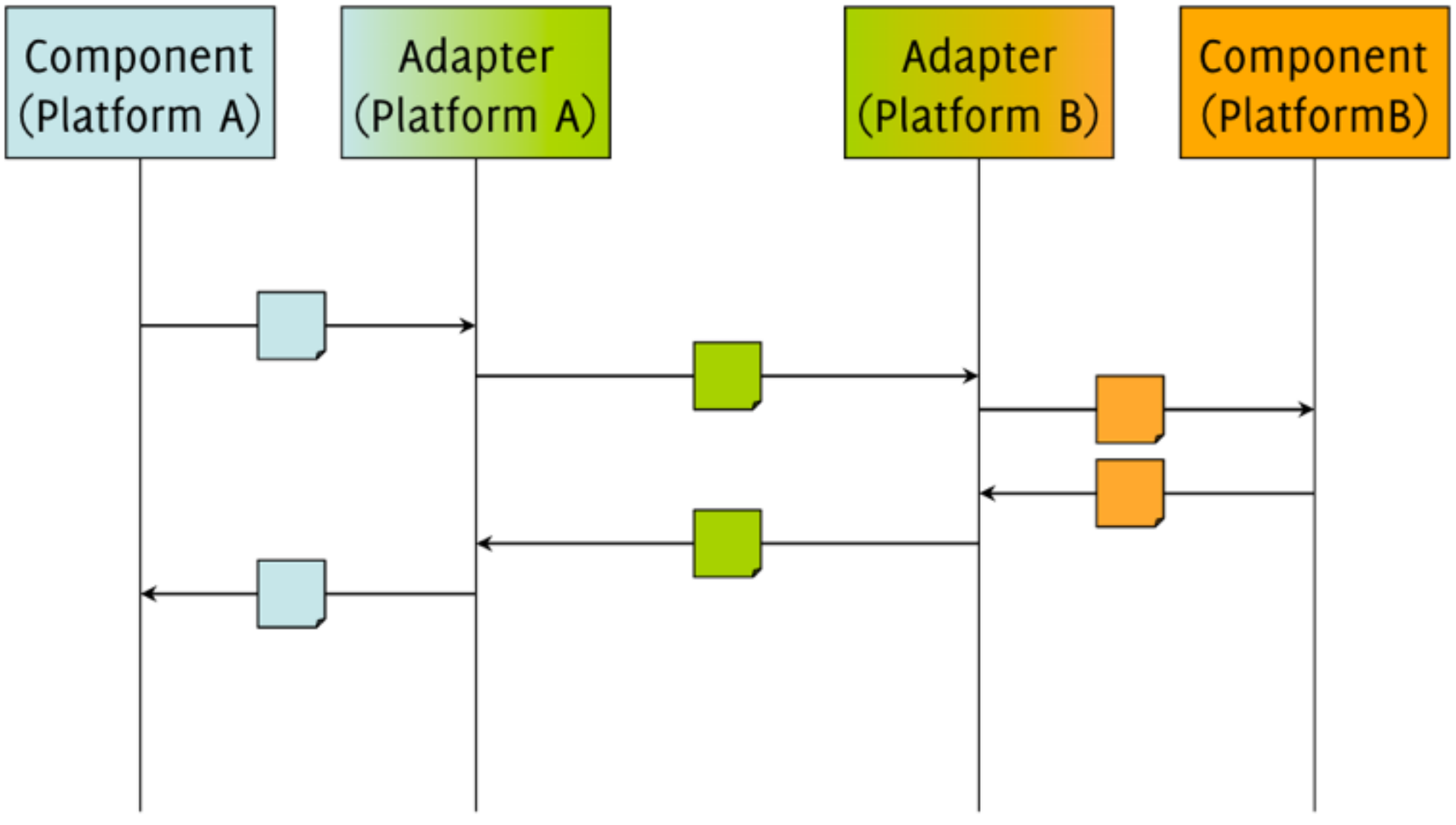
  facilitate location transparency (direct control)

- Dependency Injection

  facilitate location transparency (inversion of control)

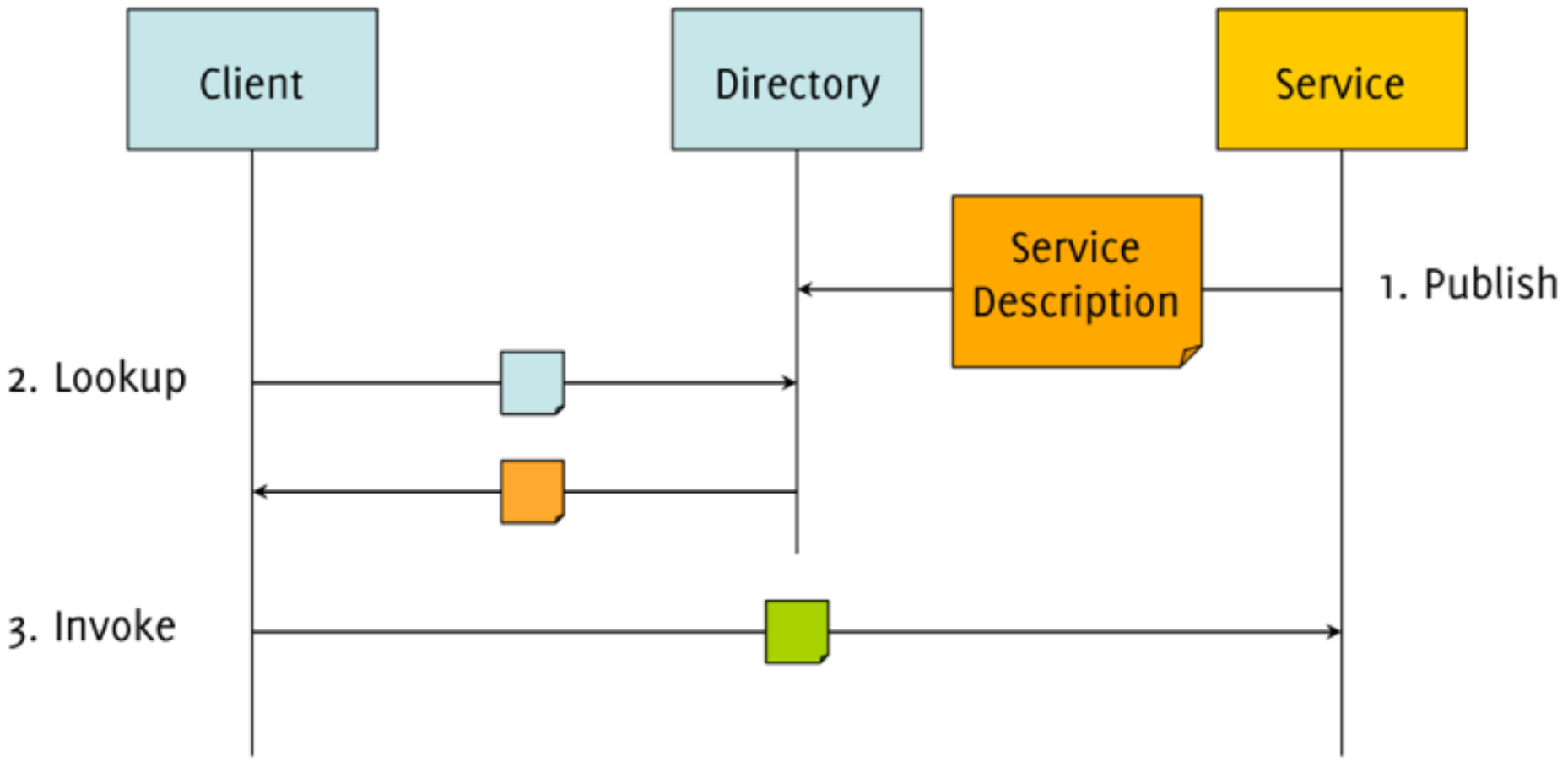
Interoperability

map to a standardized intermediate representation and communication style



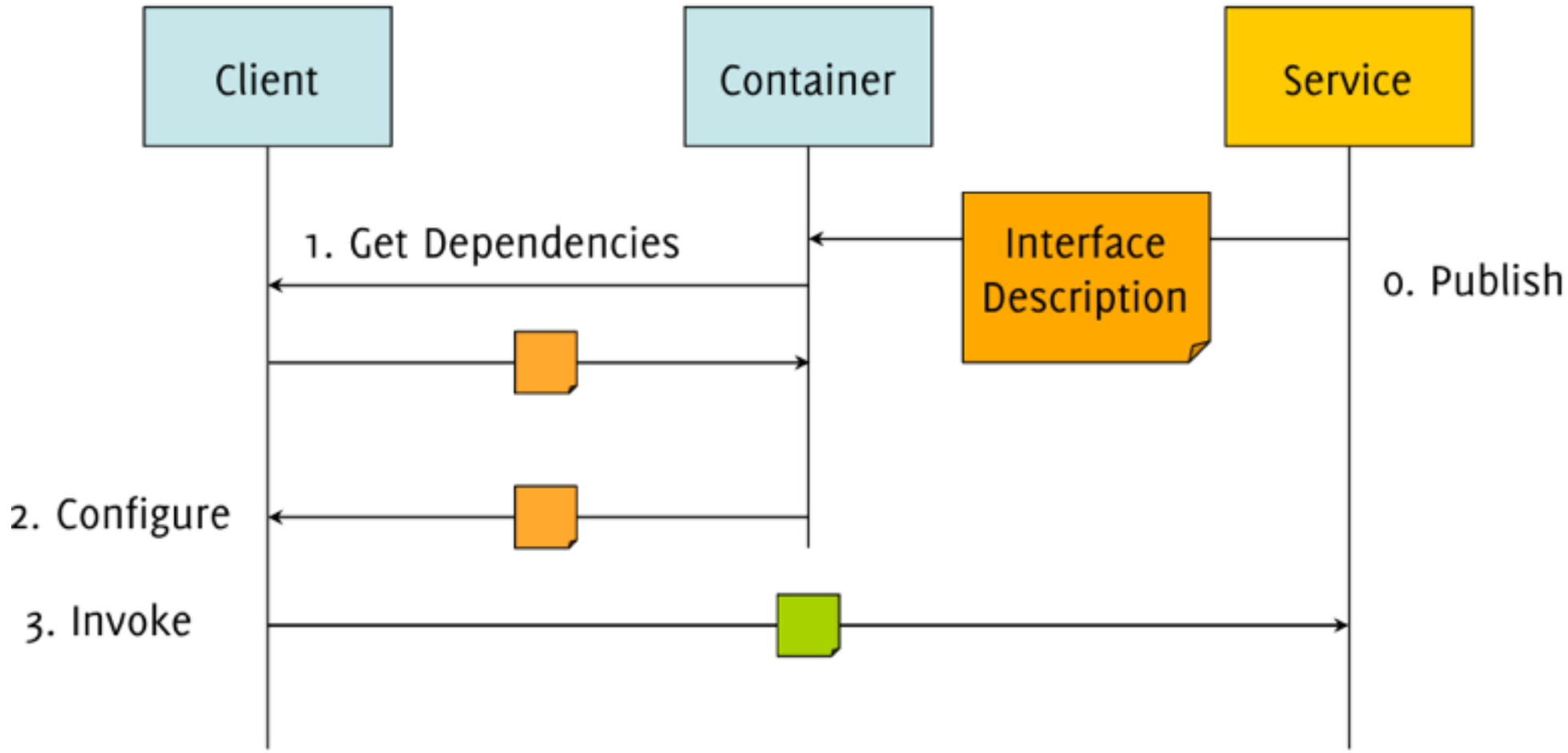
Directory

use a directory service to find service endpoints based on abstract descriptions



Dependency Injection

use a container which updates components with bindings to their dependencies



Notification Patterns

- Event Monitor

  inform clients about events happening at the service

- Observer

  promptly inform clients about state changes of a service

- Publish/Subscribe

  decouple clients from services generating events

- Messaging Bridge

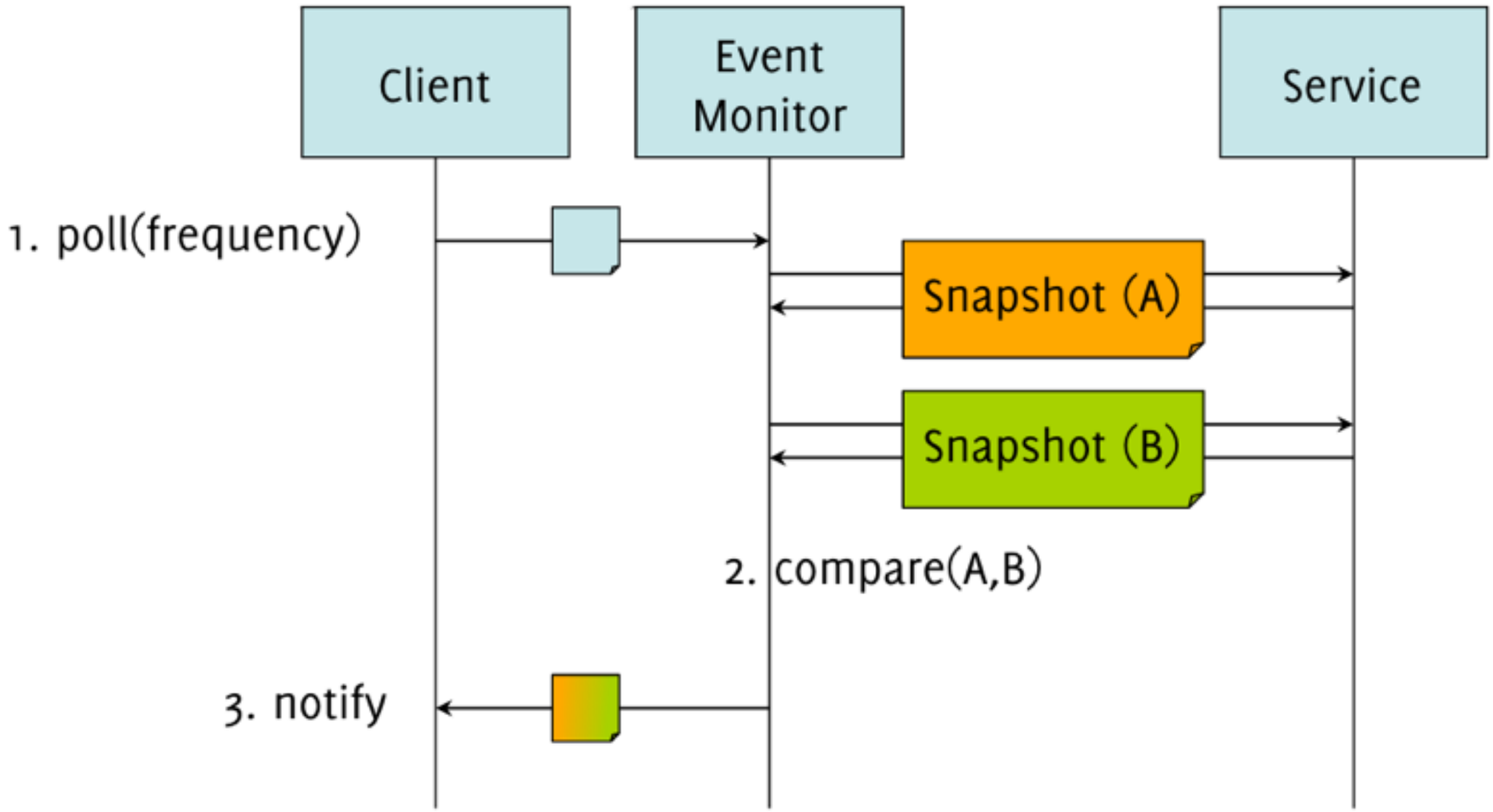
  connect multiple messaging systems

- Half Synch/Half Async

  interconnect synchronous and asynchronous components

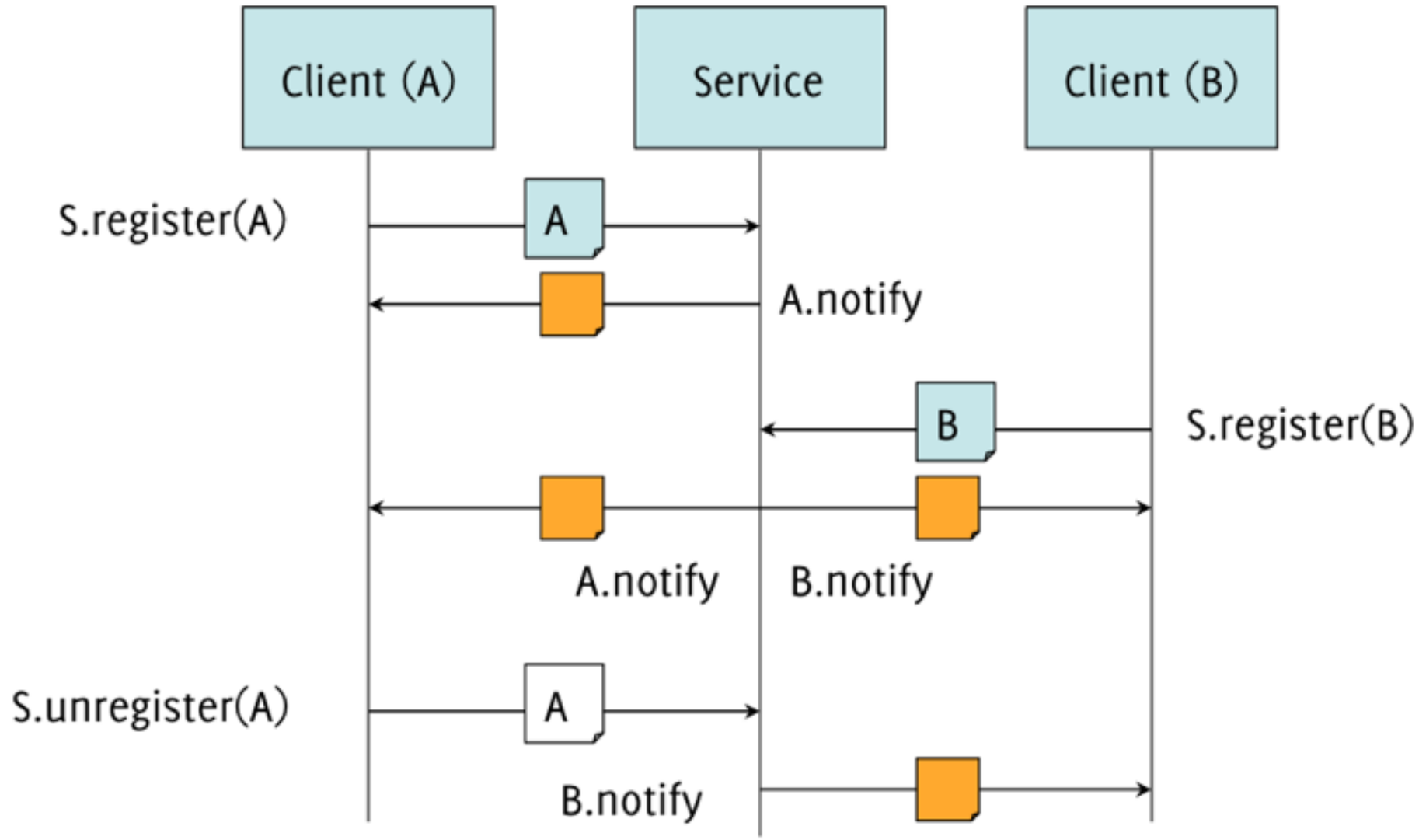
Event Monitor

poll and compare state snapshots



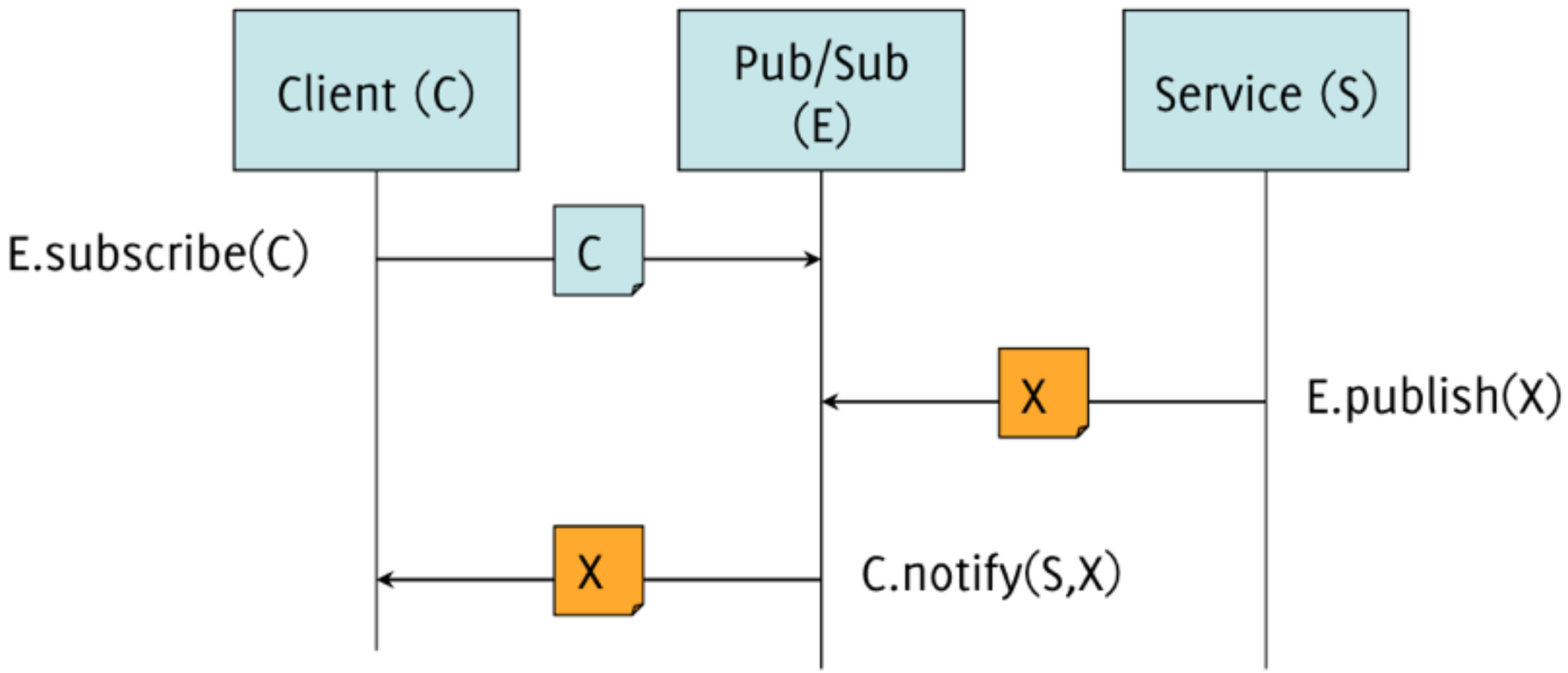
Observer

detect changes and generate events at the service



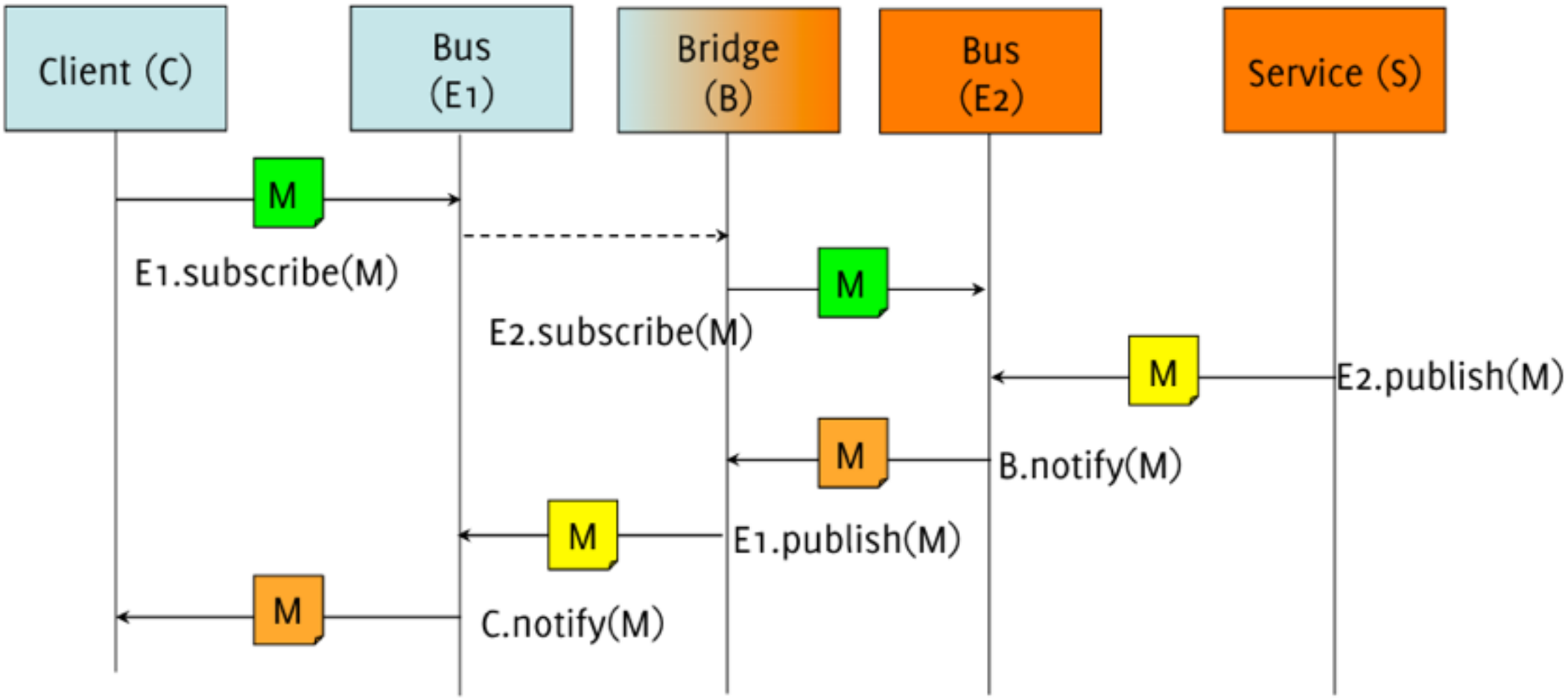
Publish/Subscribe

factor out event propagation and subscription management into a separate service



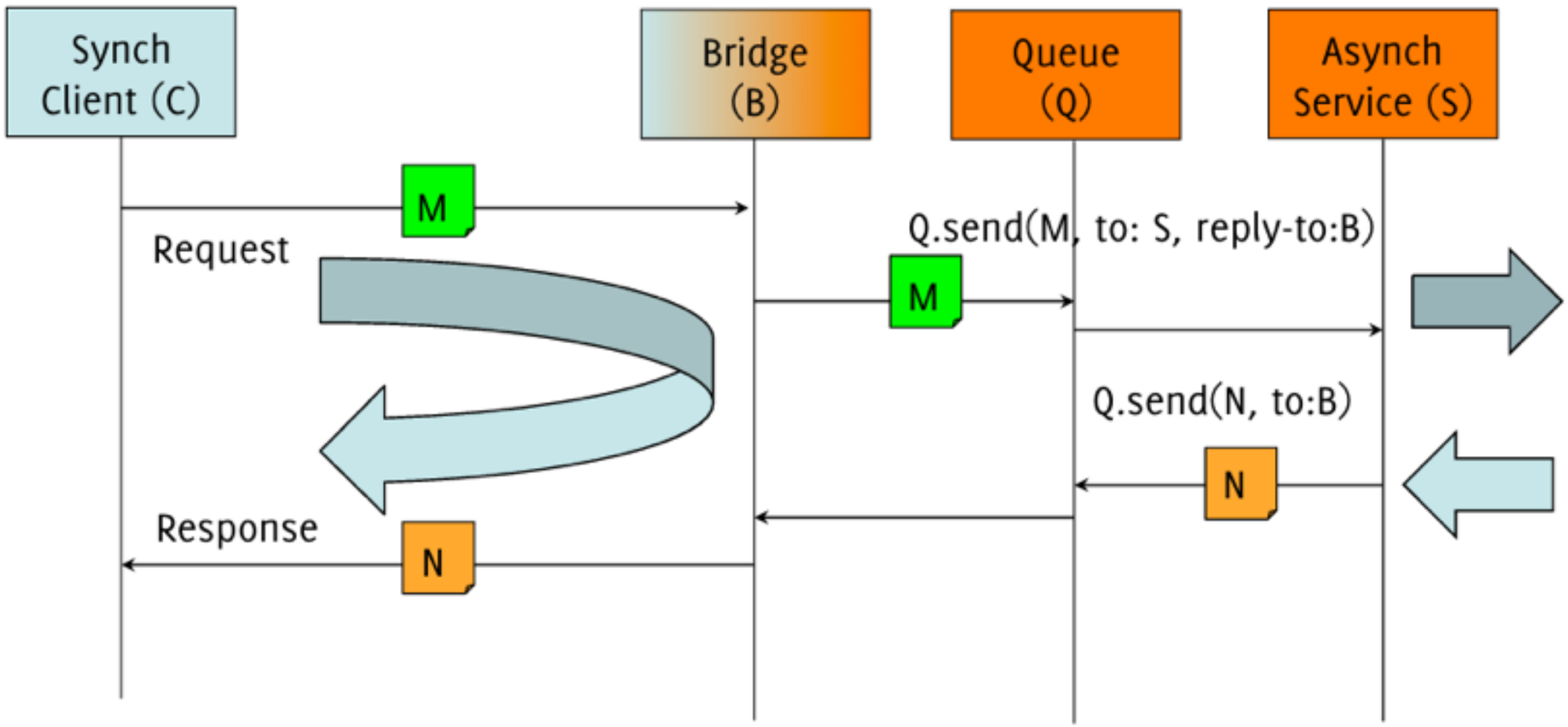
Messaging Bridge

link multiple messaging systems to make messages exchanged on one also available on the others



Half-Sync/Half-Async

Add a layer hiding asynchronous interactions behind a synchronous interface



Composition Patterns

- Scatter/Gather

  send the same message to multiple recipients which will/may reply

- Canary Call

  avoid crashing all recipients of a poisoned request

- Master/Slave

  speed up the execution of long running computations

- Load Balancing

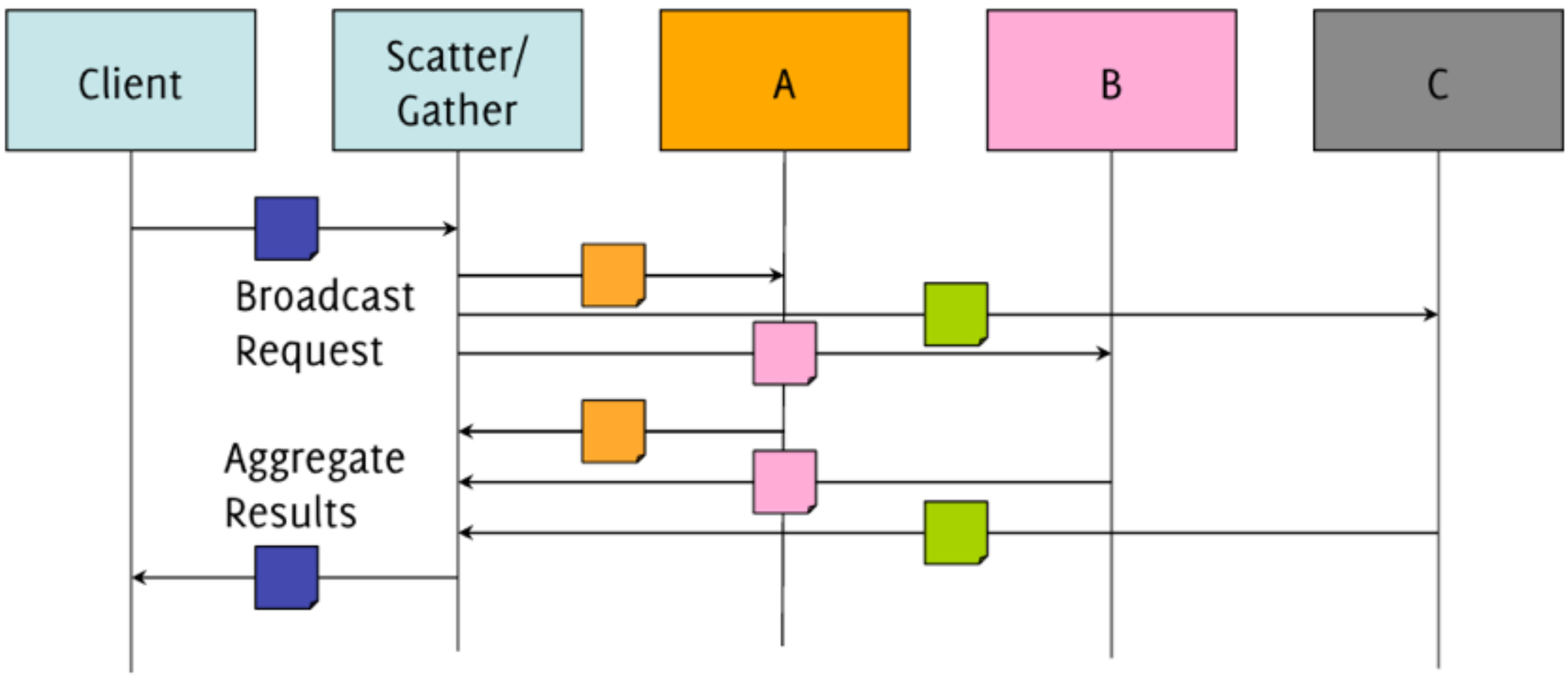
  speed up and scale up the execution of requests of many clients

- Orchestration

  improve the reuse of existing applications

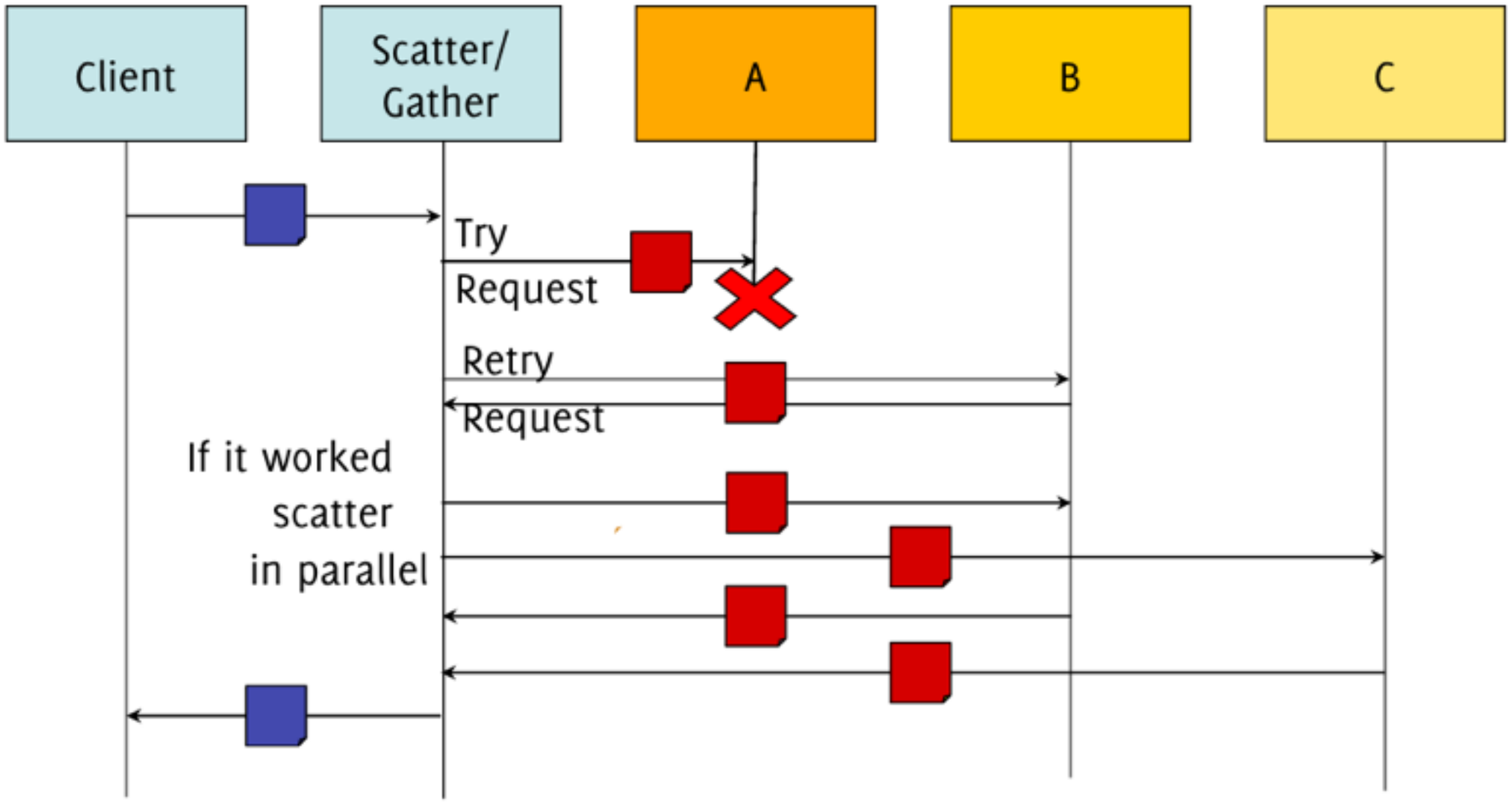
Scatter/Gather

combine the notification of the request with aggregation of replies



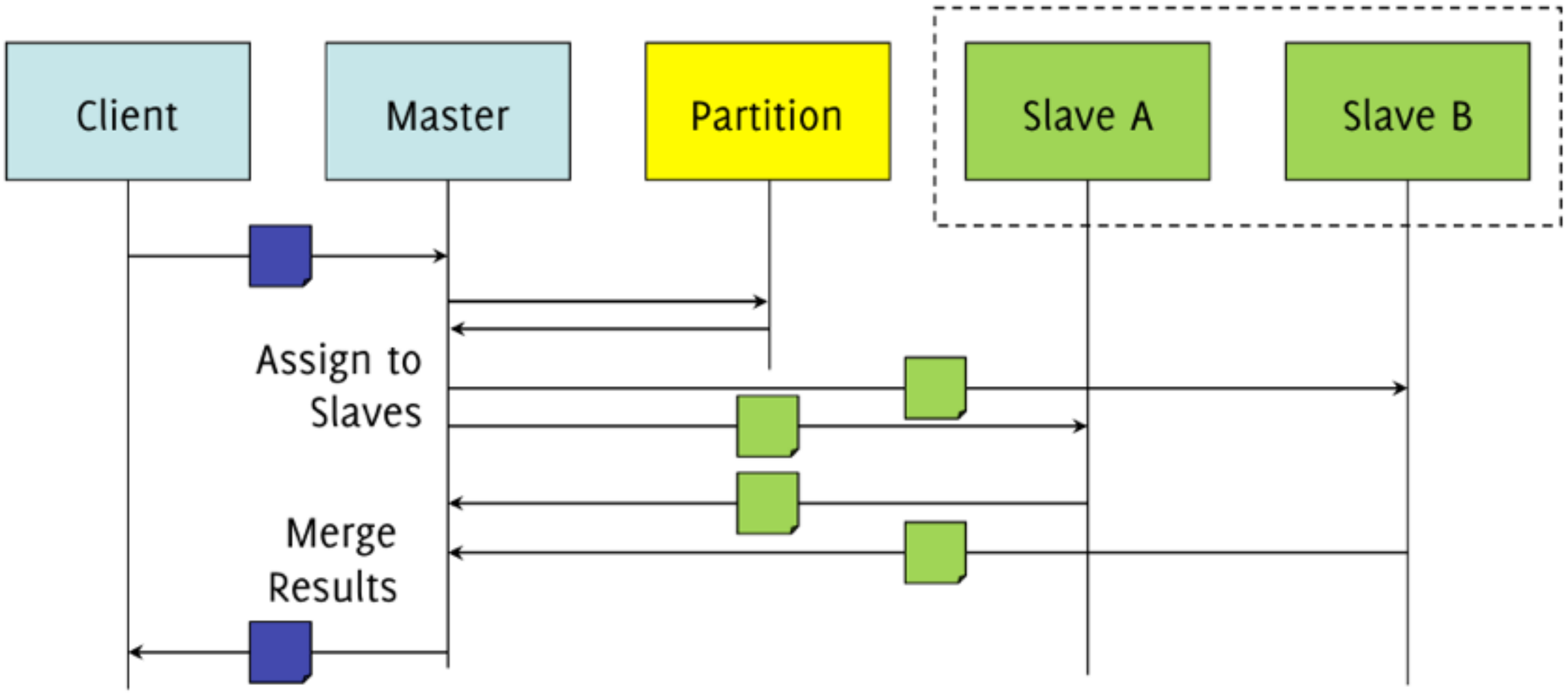
Canary Call

use an heuristic to evaluate the request



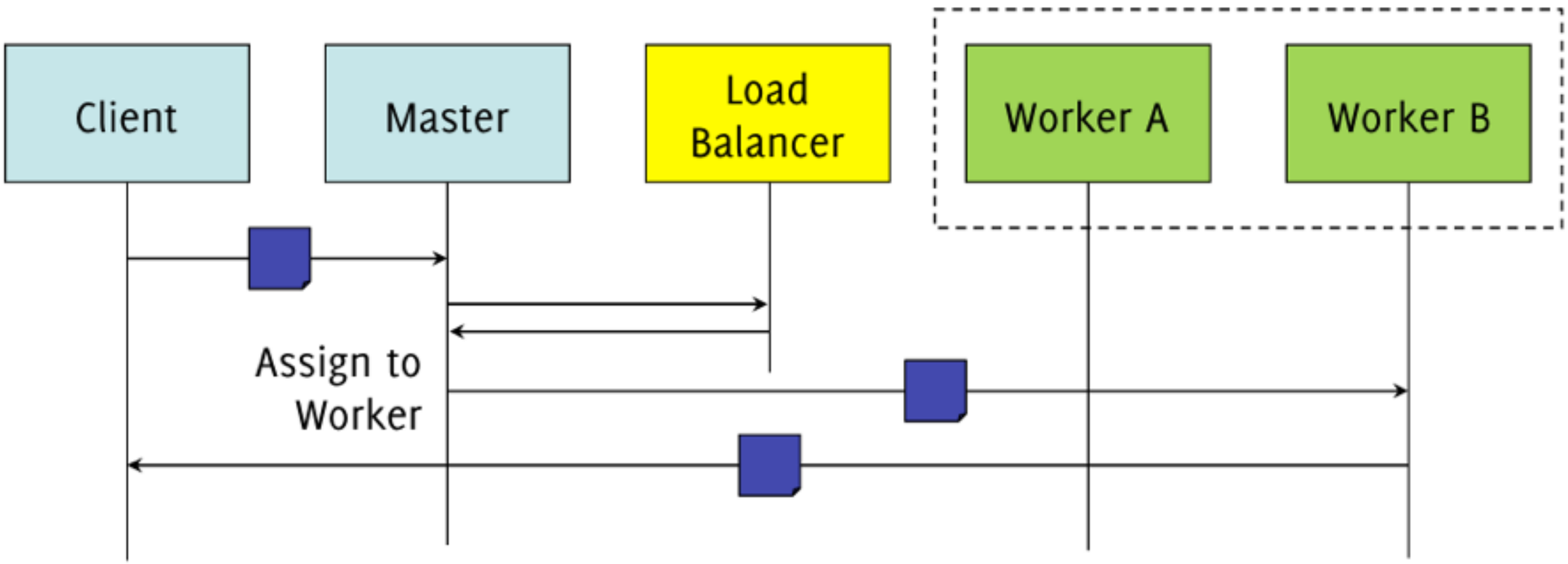
Master/Slave

split a large job into smaller independent partitions which can be processed in parallel



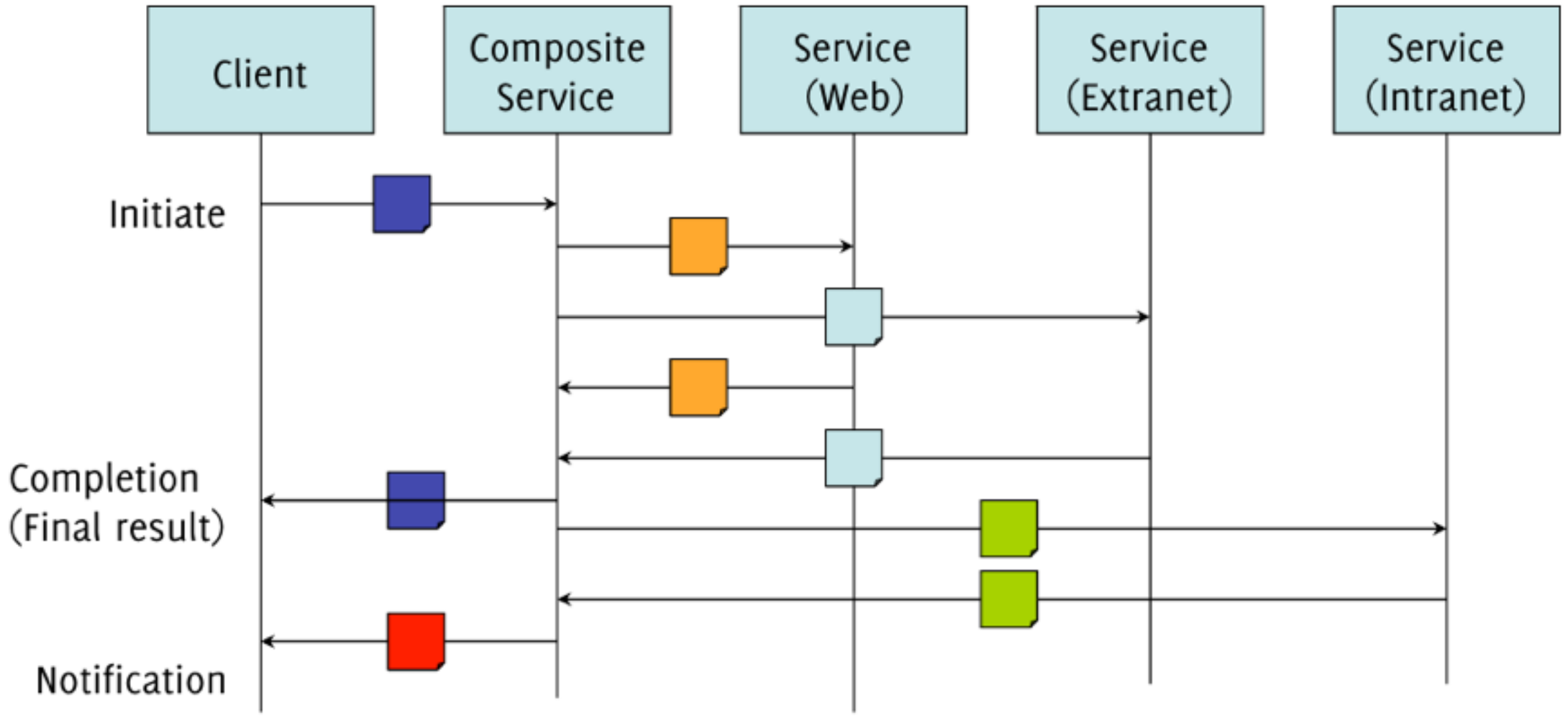
Load Balancing

deploy many replicated instances of the server on multiple machines



Composition/Orchestration

build systems out of the composition of existing ones



Software Connectors

generic building blocks

Software connectors are first-class entities, have identity, and describe all system interactions. Software connectors are application independent and orthogonal to software components.

Remote Procedure Call

- Call

Often used within the client/server architectural style and event-oriented systems as call-backs

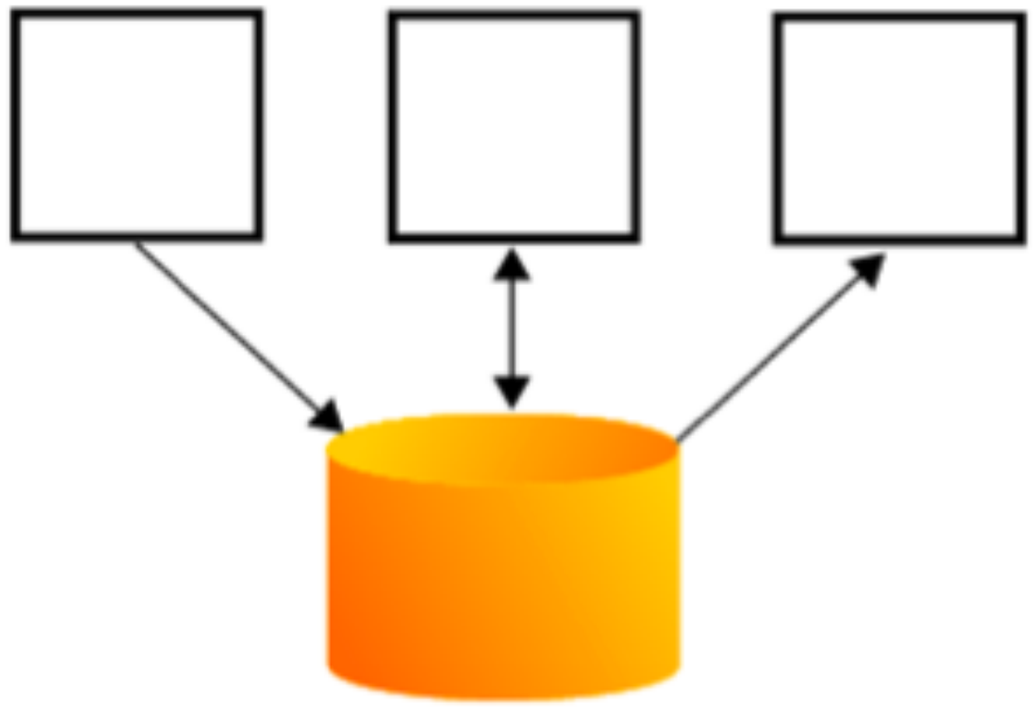
Stream

- Send
- Receive

Fits the pipe & filter architectural style

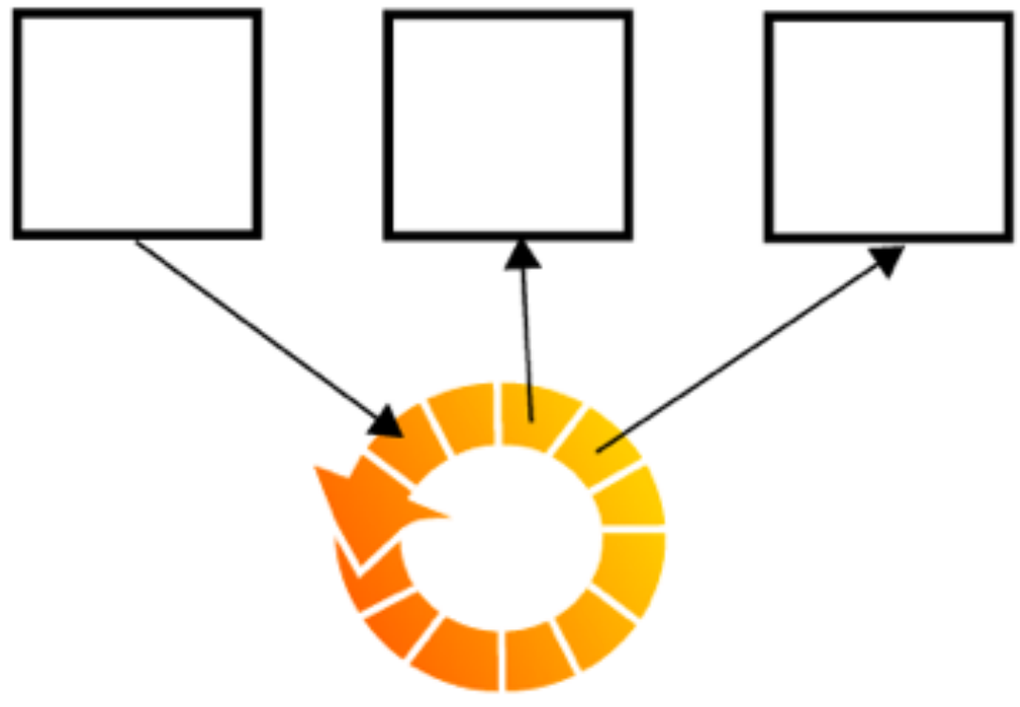
Shared Database

- Create
- Read
- Update
- Delete



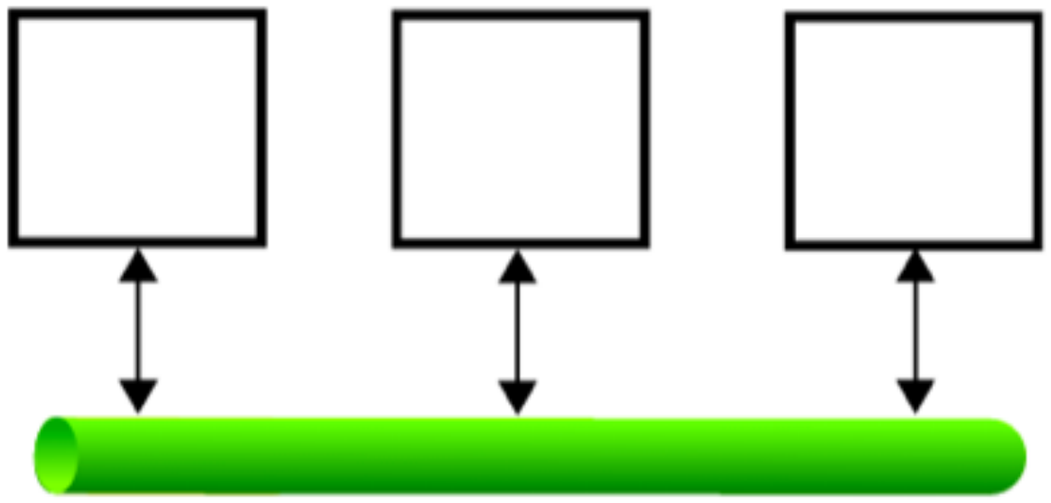
Disruptor

- Next
- Publish
- WaitFor
- Get



Message Bus

- Publish
- Subscribe
- Notify



Fits the Service Oriented style

File Transfer

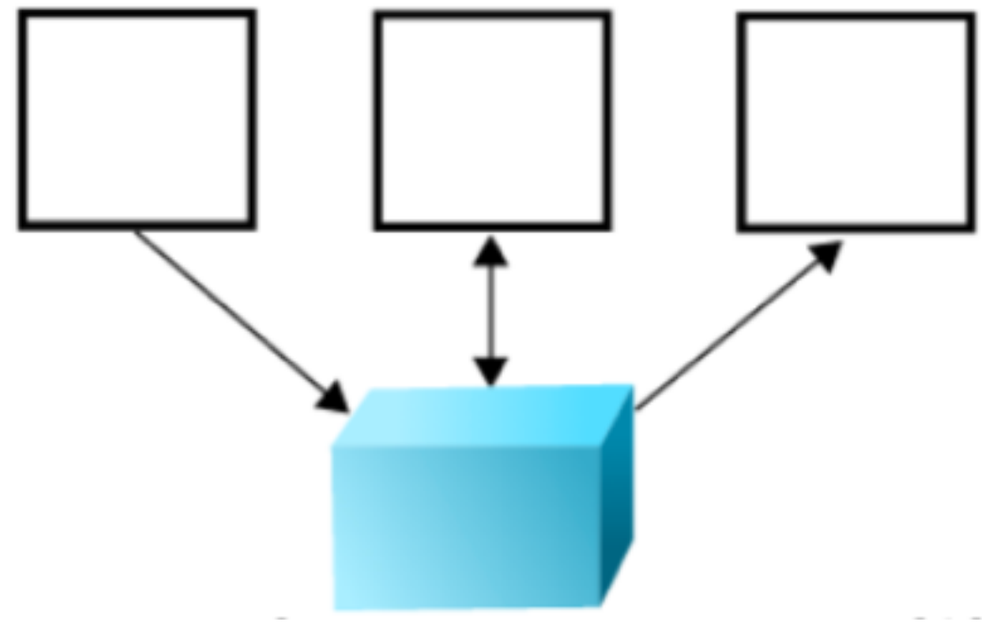
- Write
- Copy
- Watch
- Read

Linkage

- Load
- Unload
- Call
- Read/Write

Tuple Space

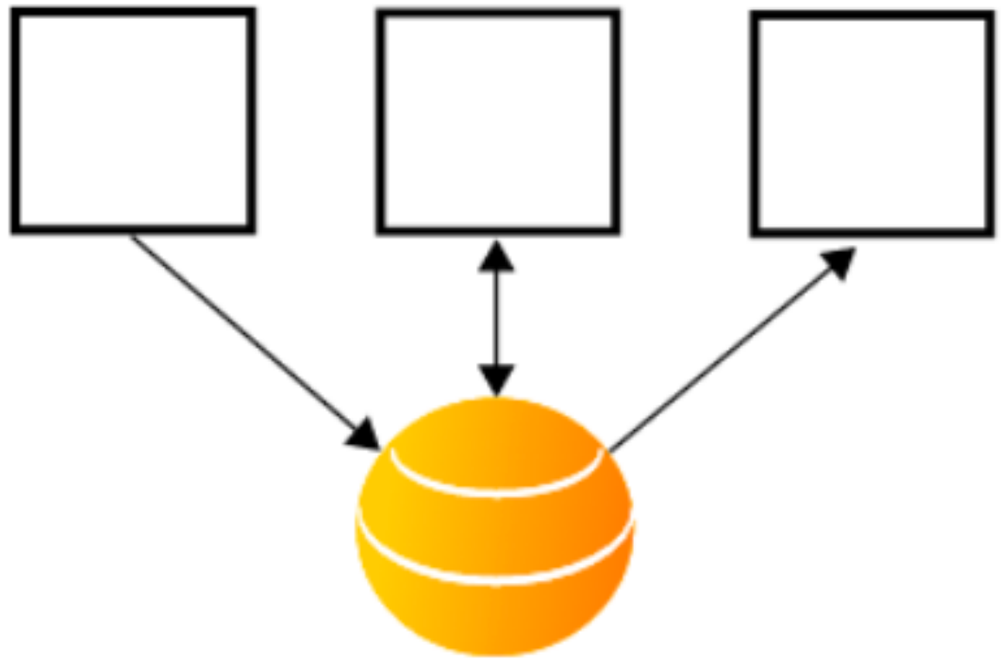
- In
- Out
- Rd



Fits the Blackboard style and the Master/Worker pattern

Web

- Get
- Put
- Post
- Delete



Fits the REST architectural style

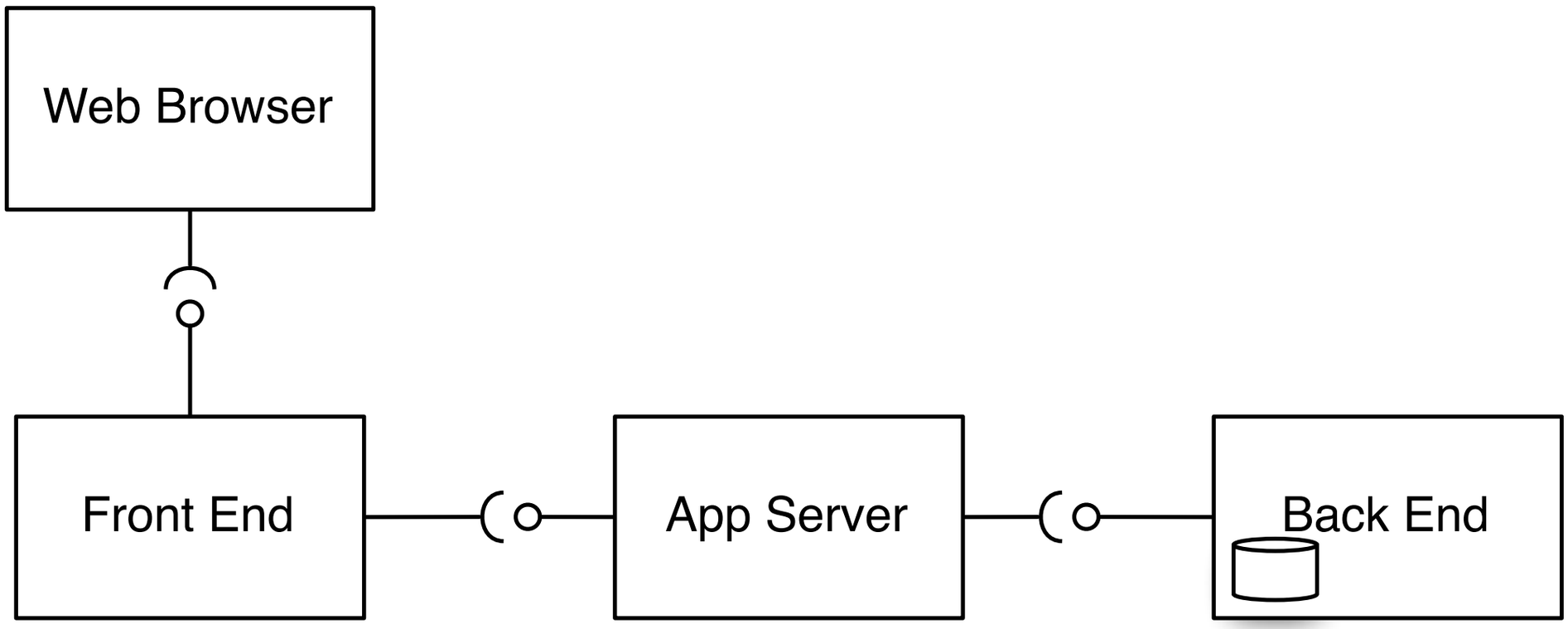
Case Study

MediaWiki

- General purpose PHP-based system for Wikis
- The core of WikiMedia project (Wikipedia)
- Long-living project (~14 years)
- In September 2014 all Wikimedia projects served ~23.2 billions of pages

Main Scenarios

- A user requests an article during normal operation and gets the rendered article HTML page.
- An editor saves an edited article during normal operation and the article is saved.

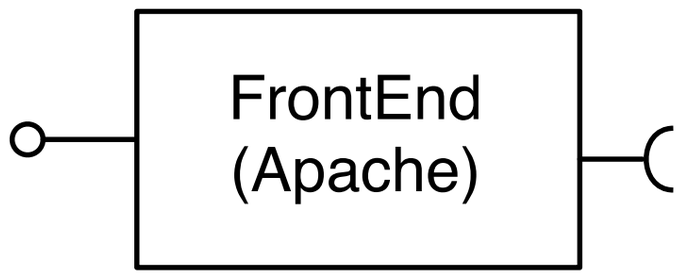

Web Browser
Front End
App Server
Back End

Performance Tactics

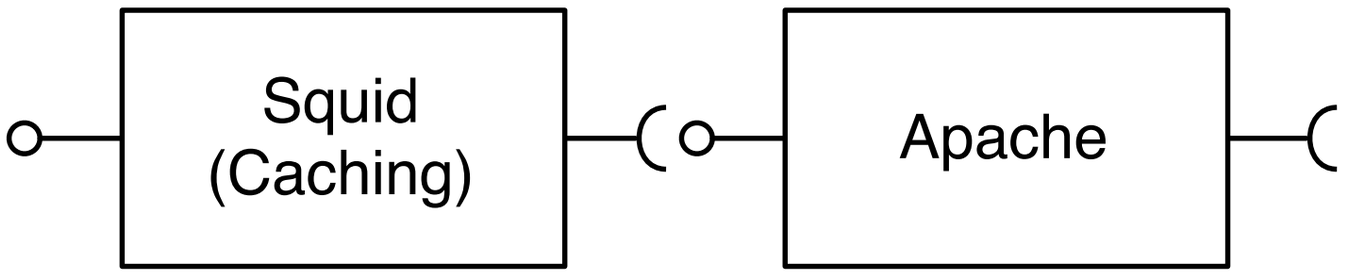
- Control Resource Demand
  - *Increase the resource efficiency (caching)*
  - *Prioritize events (deferred article updates)*
  - *Reduce overhead (precompile PHP and HTML)*

- Manage Resources
  - *Introduce concurrency (Distributed database)*
  - *Schedule resources (Load balancer)*
  - *Multiple copies of data and computations*

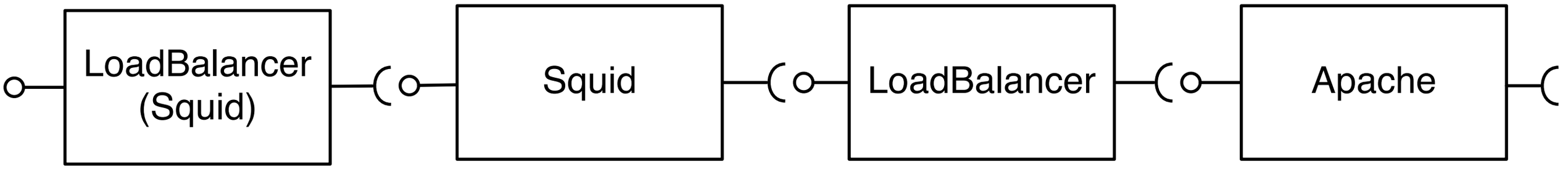
Caching + Load Balancing



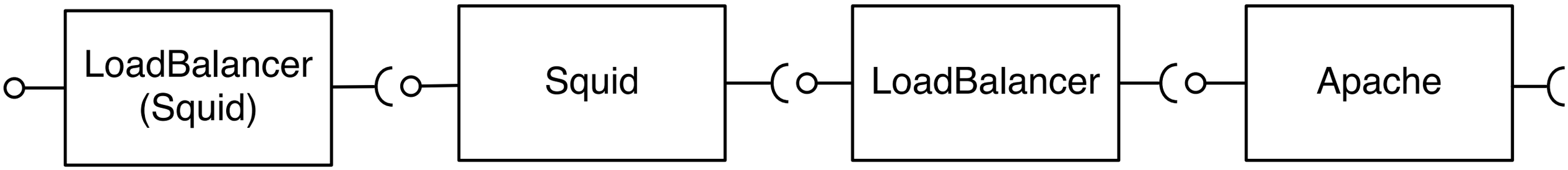
Caching + Load Balancing

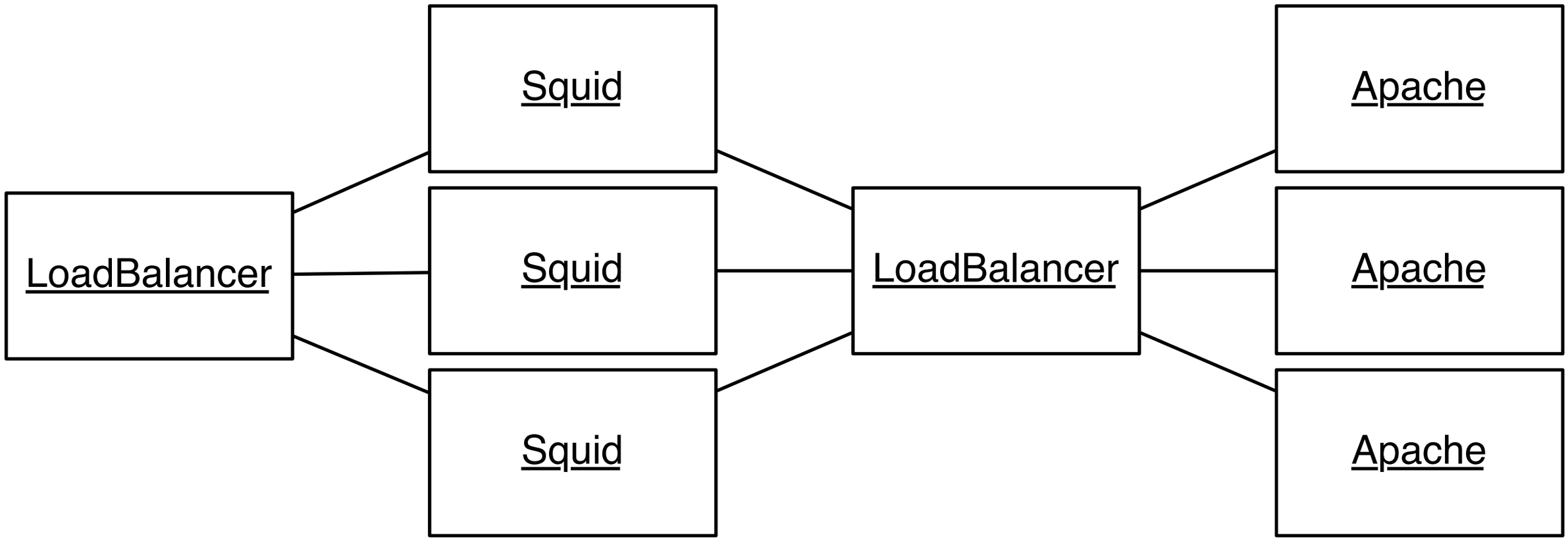


Caching + Load Balancing

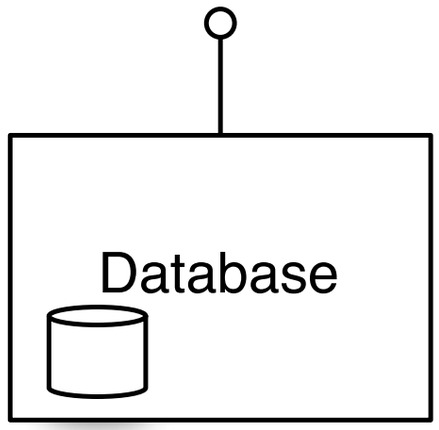


Caching + Load Balancing

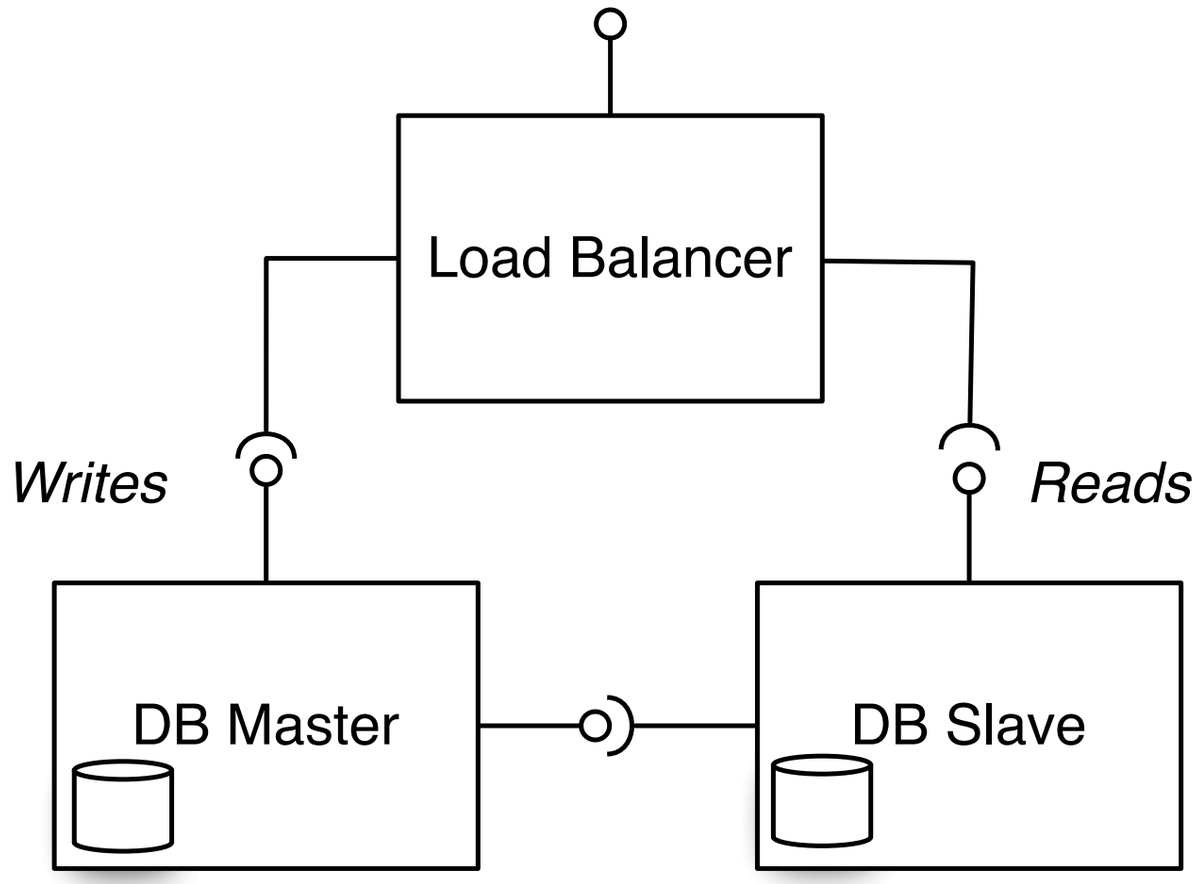




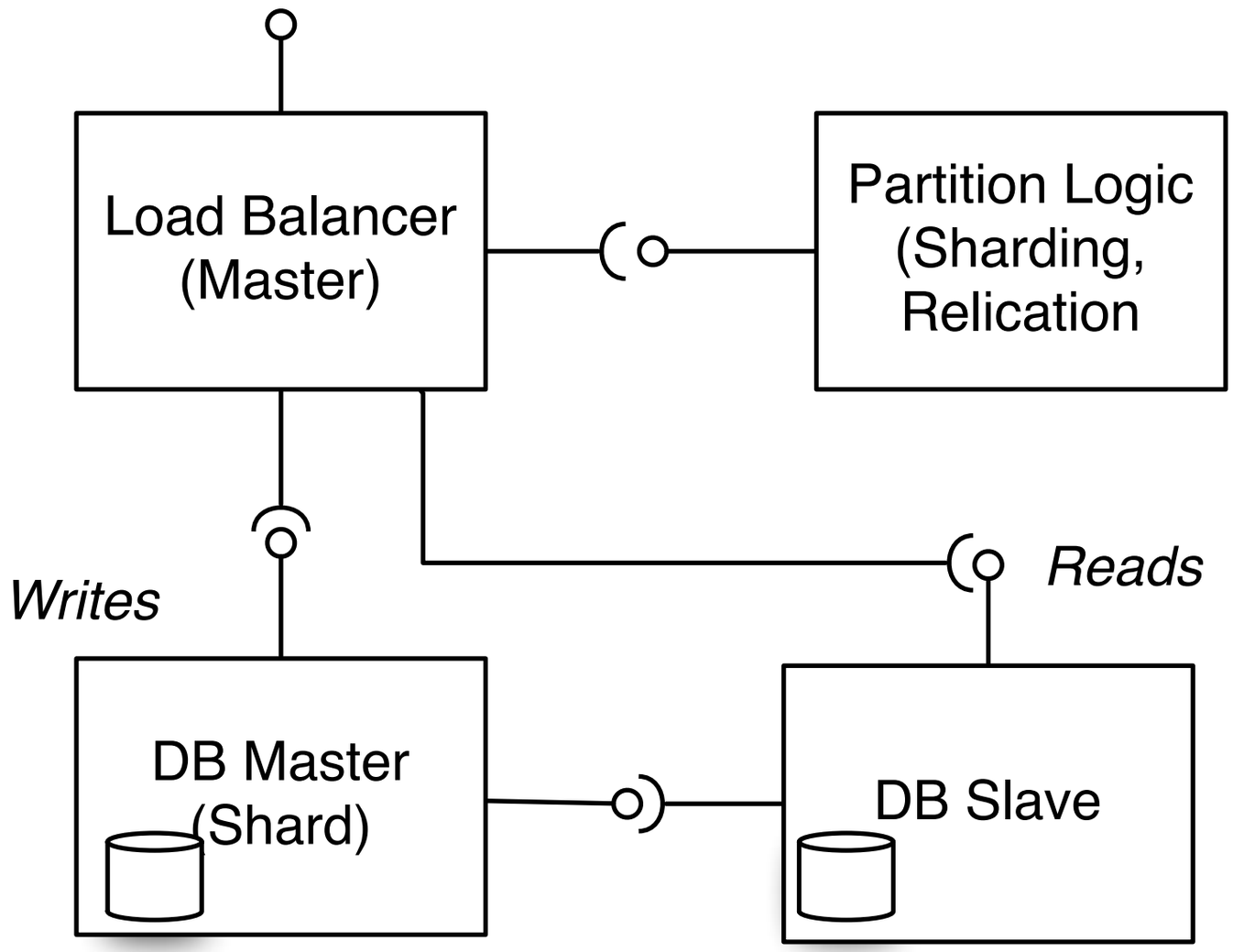
Distribution + Replication

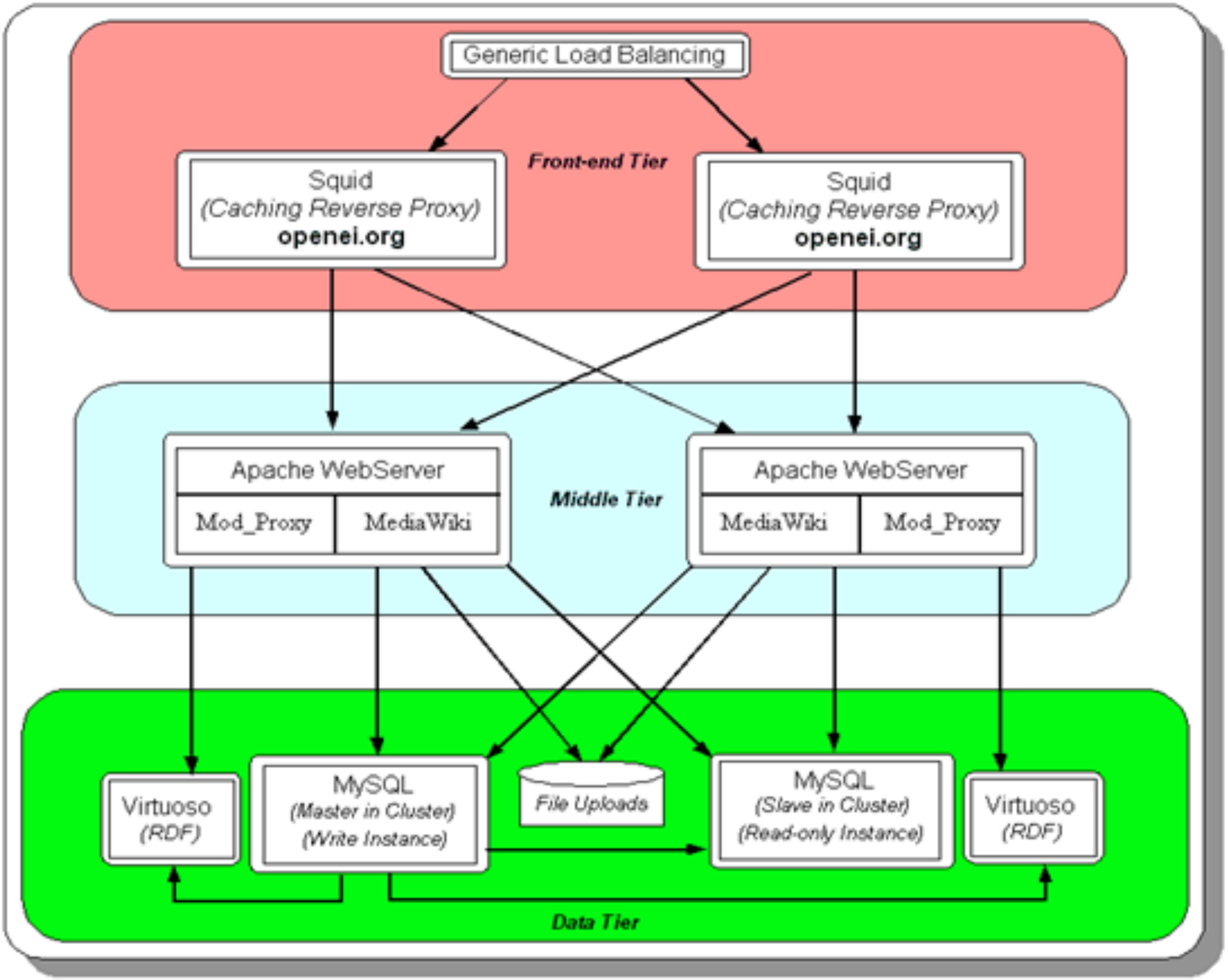


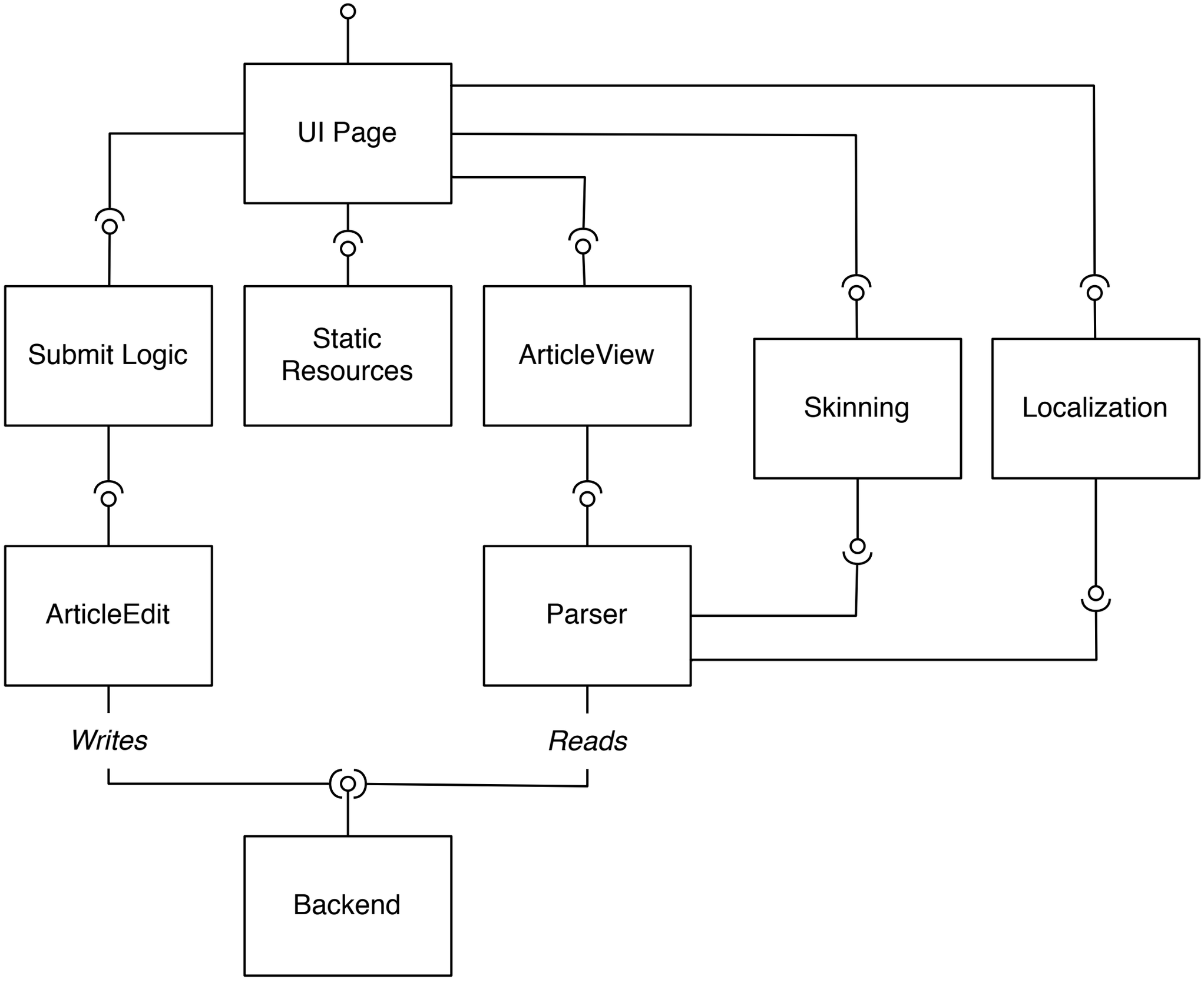
Distribution + Replication

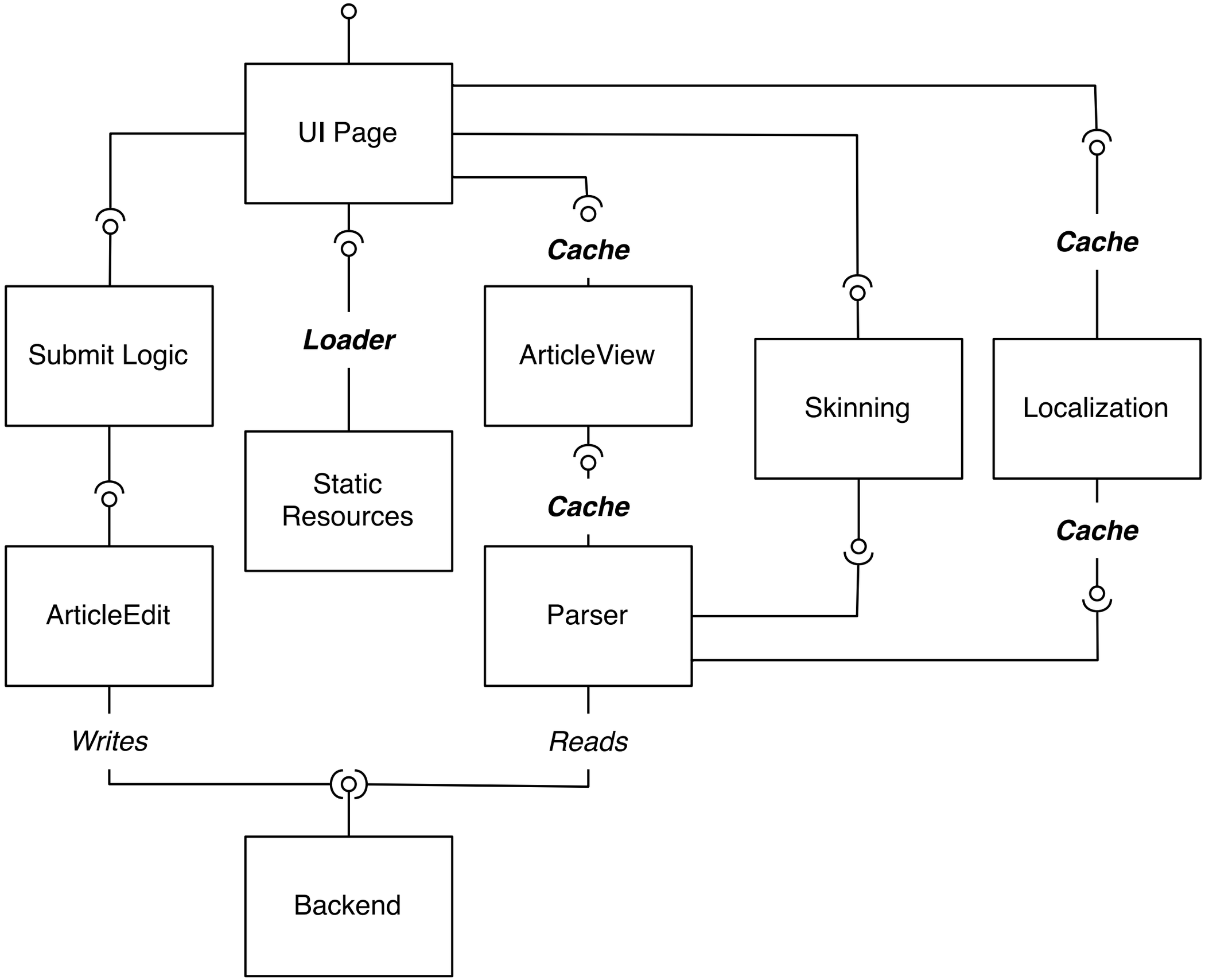


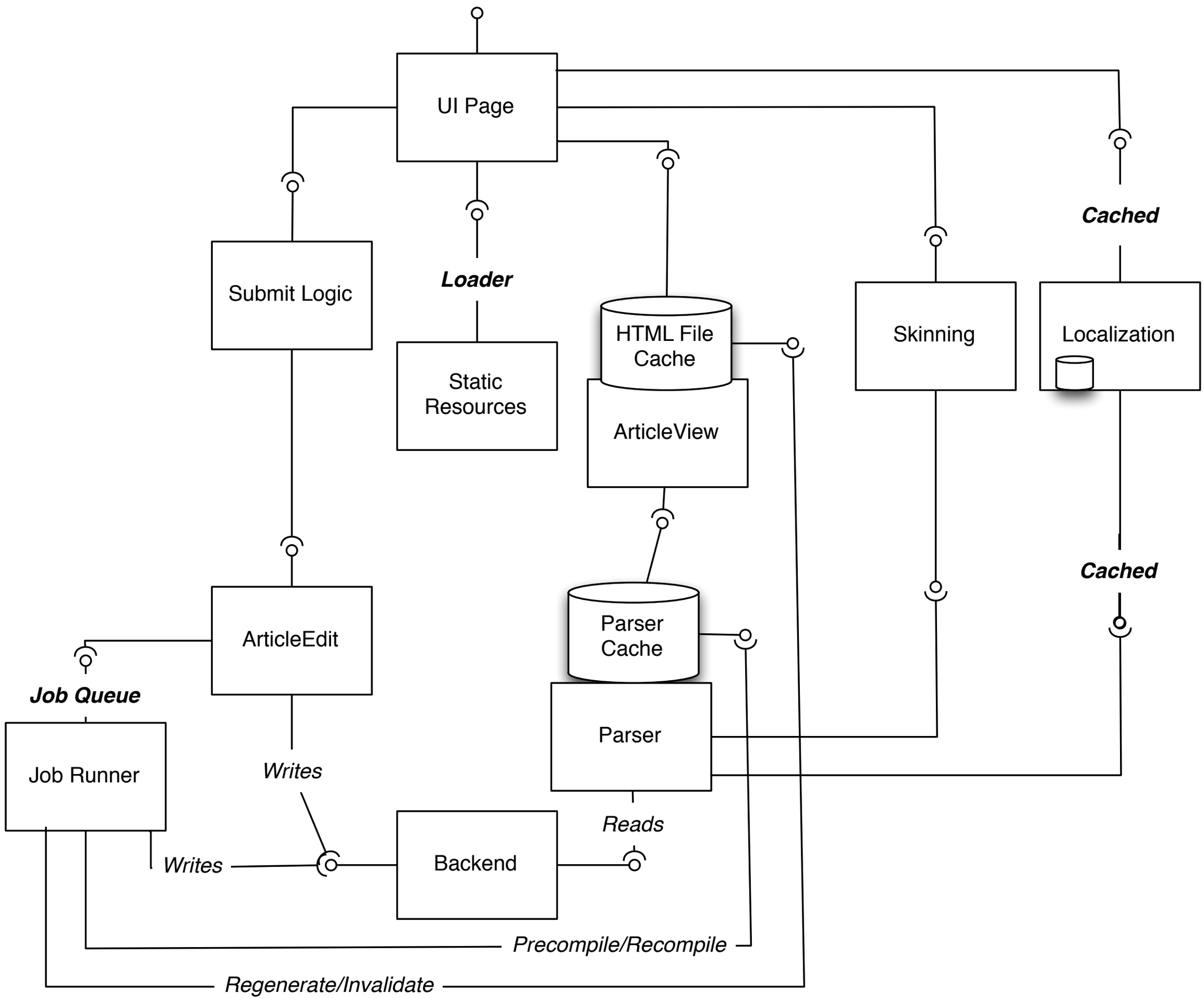
Distribution + Replication




Generic Load Balancing
Front-end Tier
Squid
(Caching Reverse Proxy)
openei.org
Squid
(Caching Reverse Proxy)
openei.org
Apache WebServer
Mod_Proxy
MediaWiki
Middle Tier
Apache WebServer
MediaWiki
Mod_Proxy
Virtuoso
(RDF)
MySQL
(Master in Cluster)
(Write Instance)
File Uploads
MySQL
(Slave in Cluster)
(Read-only Instance)
Virtuoso
(RDF)
Data Tier


UI Page
Submit Logic
Static Resources
ArticleView
Skinning
Localization
ArticleEdit
Parser
Writes
Reads
Backend


UI Page
Submit Logic
Loader
Cache
ArticleView
Skinning
Cache
Localization
Static Resources
Cache
Cache
ArticleEdit
Parser
Writes
Reads
Backend

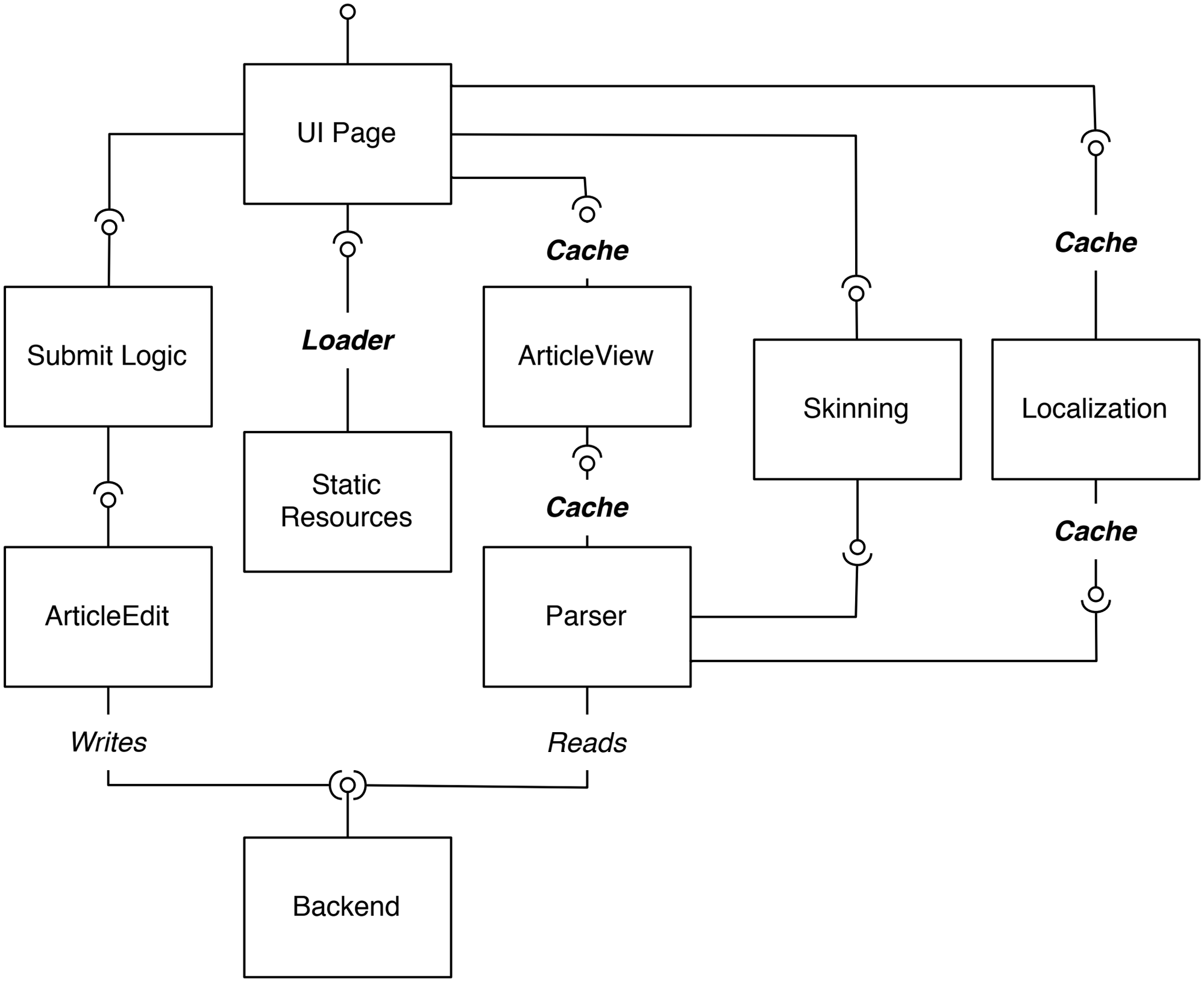

UI Page
Submit Logic
Loader
Static Resources
HTML File Cache
ArticleView
Skinning
Localization
Cached
Cached
ArticleEdit
Job Queue
Job Runner
Writes
Writes
Backend
Parser Cache
Parser
Reads
Precompile/Recompile
Regenerate/Invalidate

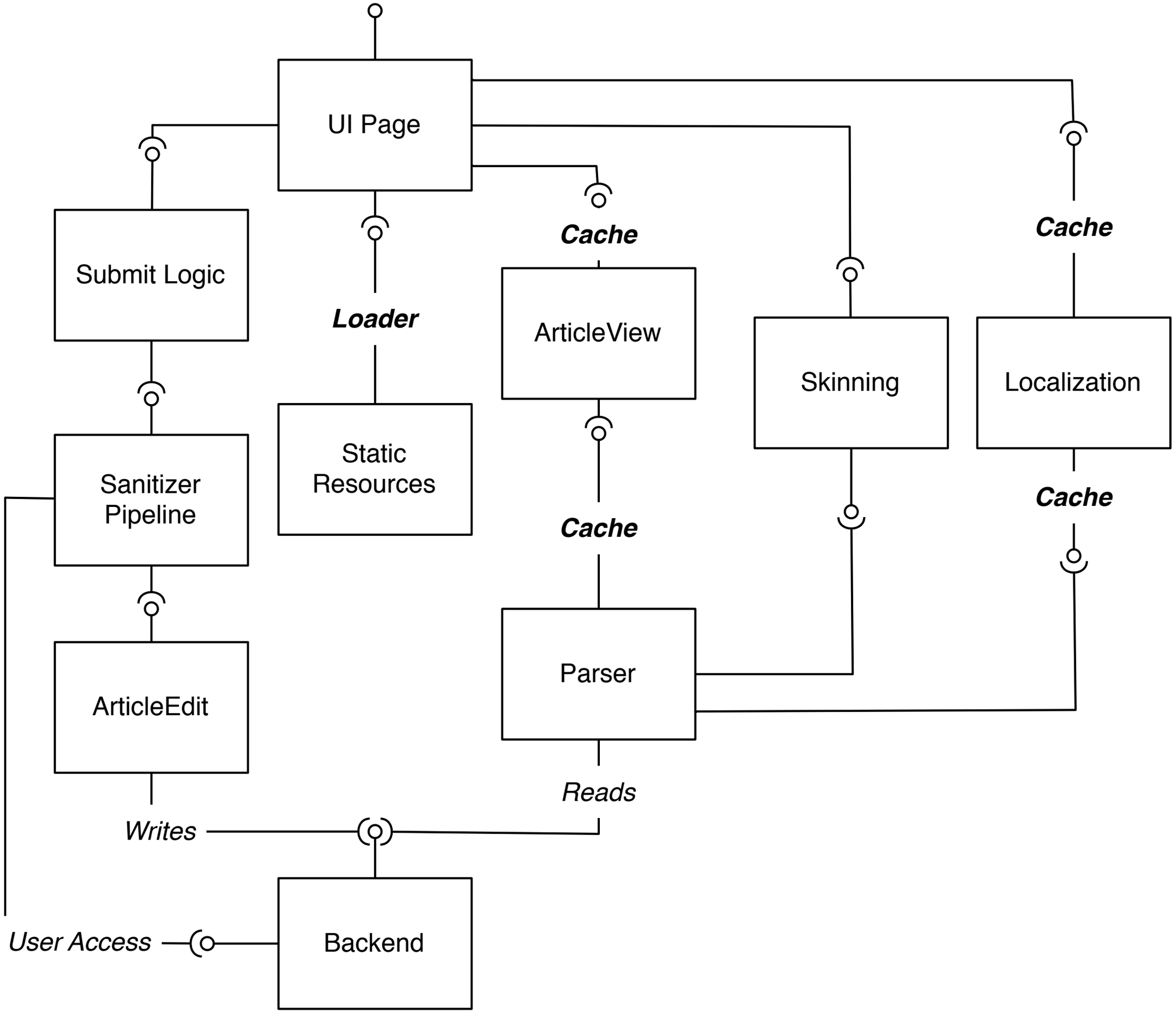
Security/Availability Tactics

- Prevent Attacks
  - *Challenge Tokens (CSRF)*
  - *Validation (User) and Sanitization (SQL Injection, XSS)*

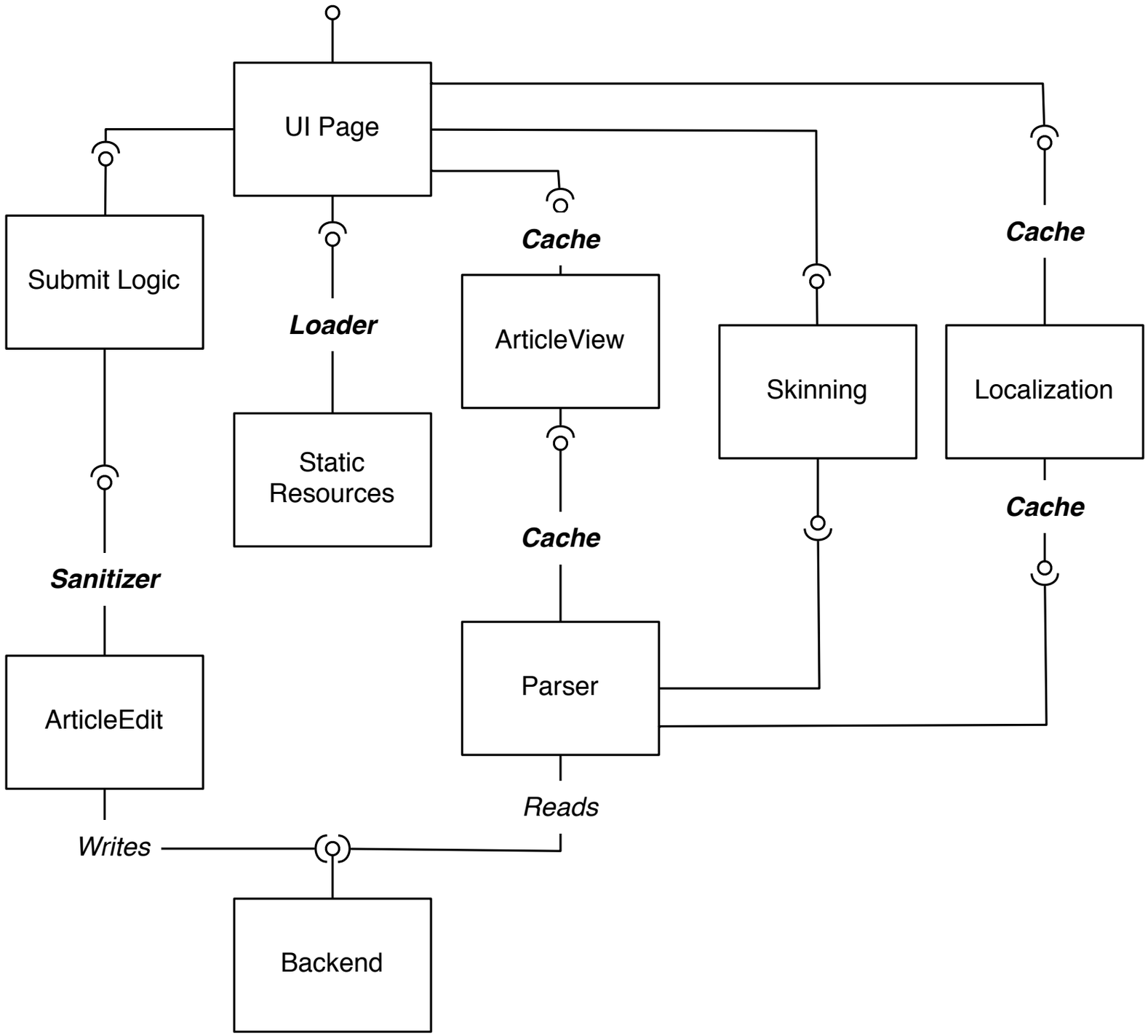
- Resist Attacks
  - *Maintain multiple copies of computations.*
  - *Maintain multiple copies of data*

- Recover from Attacks
  - *DB Versioning (Recovery from data loss)*

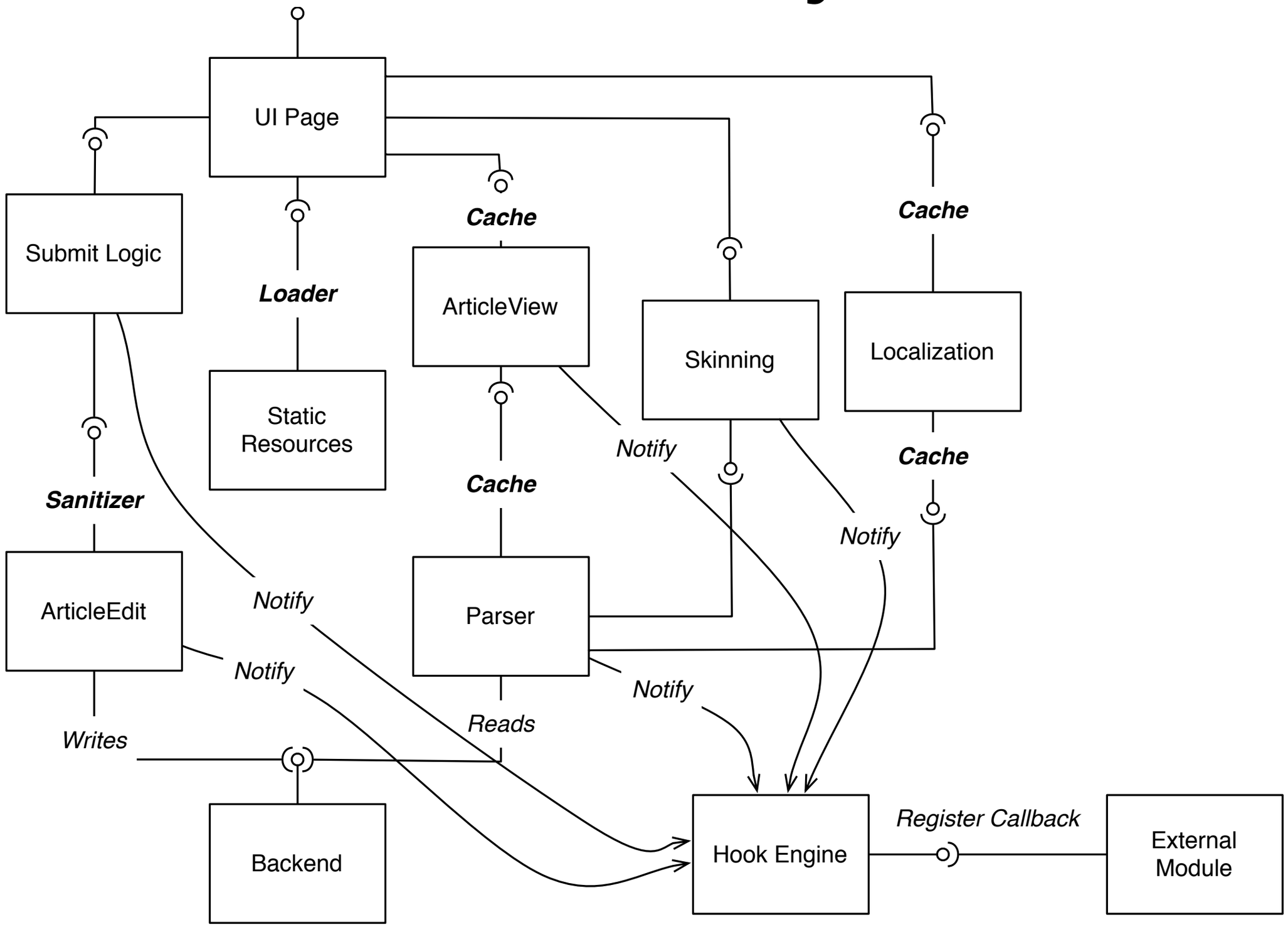

UI Page
Submit Logic
Loader
Cache
ArticleView
Skinning
Cache
Localization
Static Resources
Cache
Cache
ArticleEdit
Parser
Writes
Reads
Backend


UI Page
Submit Logic
Loader
Cache
Cache
Sanitizer Pipeline
Static Resources
ArticleView
Skinning
Localization
Cache
Cache
ArticleEdit
Parser
Reads
Writes
User Access
Backend

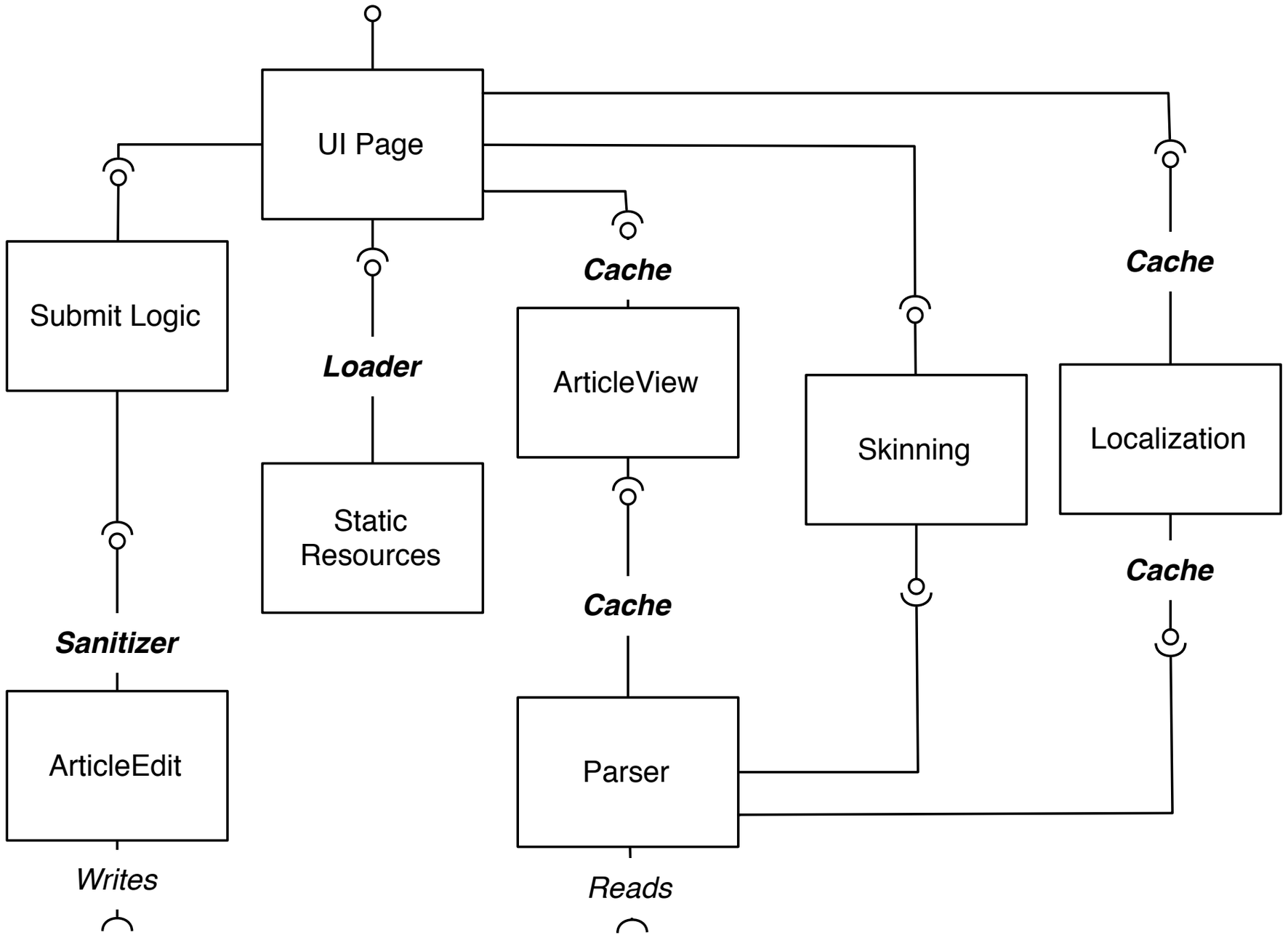
Extensibility



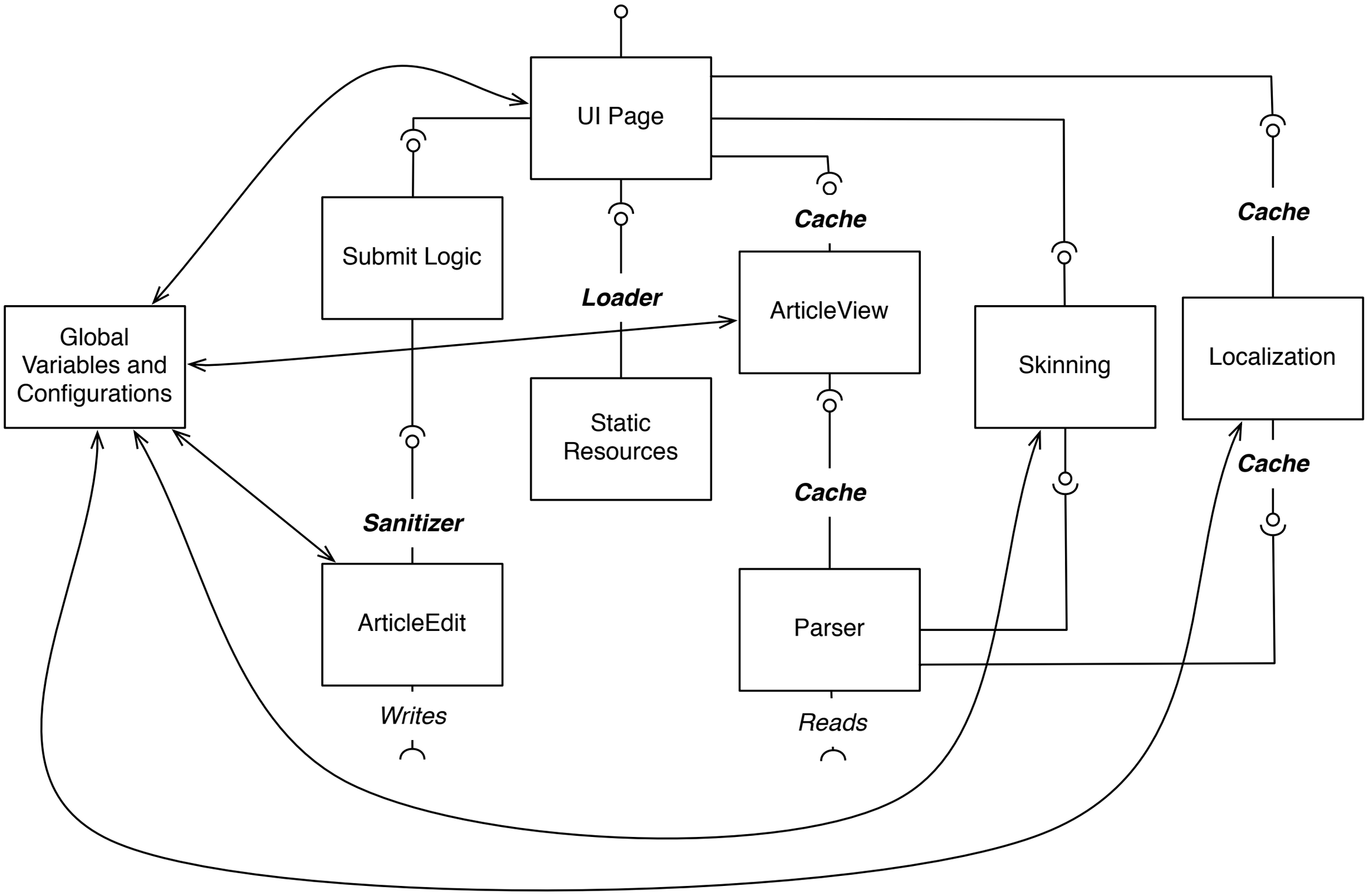
Extensibility



Configurability/Customizability



Configurability/Customizability



Architectural Styles

“Looking at the source code, it becomes evident that MediaWiki’s development process did not focus on strictly following textbook architectural styles and patterns.”

- Layered
  - *FrontEnd/Network, Application, Backend/Database*
  - *Multi-level caching*
- Blackboard
  - *Global variables*

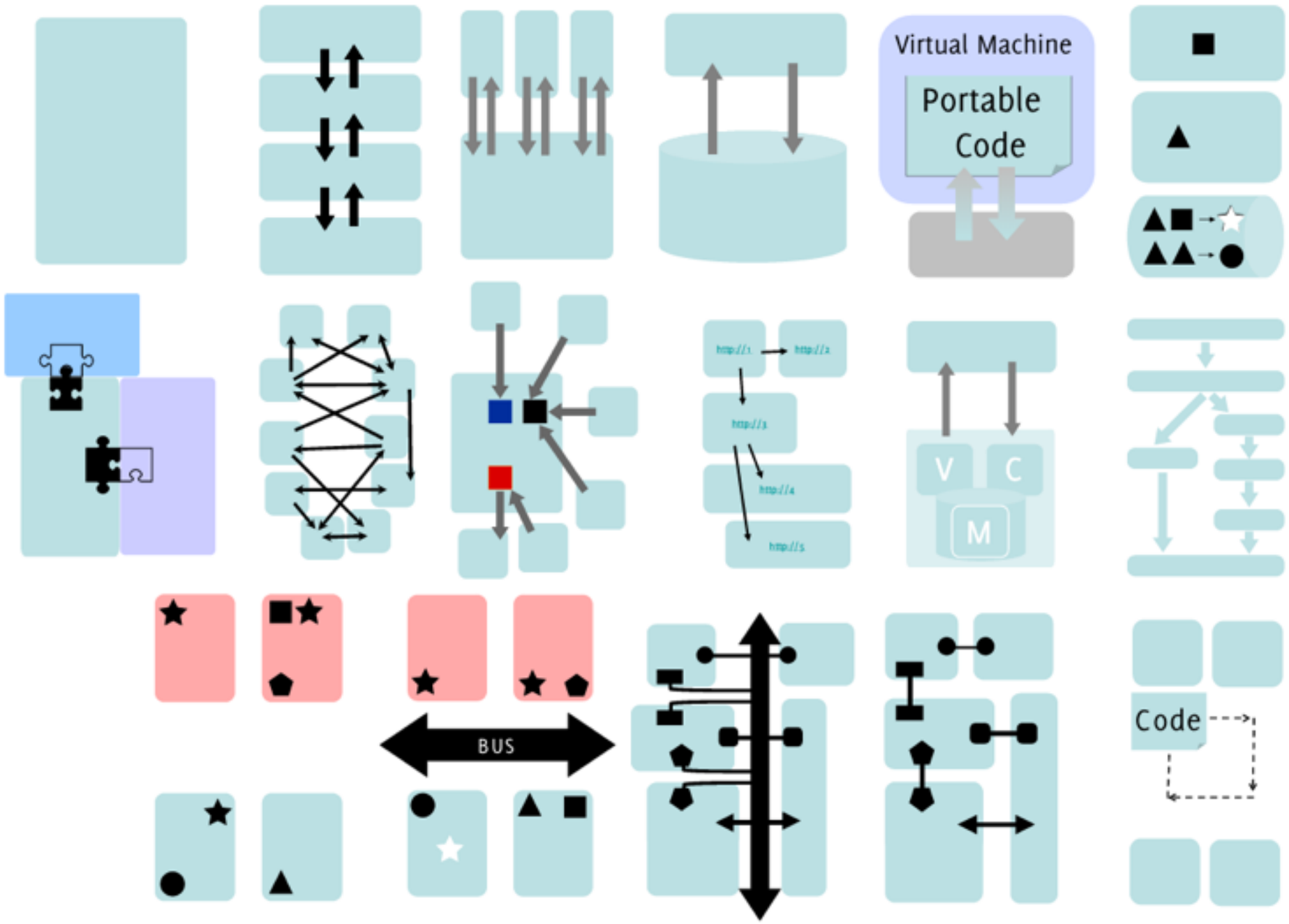
Architectural Patterns

- Presenter view (skinning)
- Publish/Subscribe (hooks)
- Master/Slave (database)
- Load balancer (network,database)
- Event monitor (cache invalidation)

Cheat Sheets

Architectural Styles

Layered
Client/Server
Data-Centric
Virtual Machine
Rule Based
Plugin
Peer to Peer
REST
Rails
Pipe and Filter
Event-Driven
Publish/Subscribe
Service Oriented
Component Based
Mobile Code
Blackboard



Layered Patterns

- State-Logic-Display

  separate elements with different rate of change

- Model-View-Controller

  support many interaction and display modes for the same content

- Presenter-View

  keep a consistent look and feel across a complex UI

Component Patterns

- Interoperability

  enable communication between different platforms

- Directory

  facilitate location transparency

- Dependency Injection

  facilitate location transparency

Notification Patterns

- Event Monitor

  inform clients about events happening at the service

- Observer

  promptly inform clients about state changes of a service

- Publish/Subscribe

  decouple clients from services generating events

- Messaging Bridge

  connect multiple messaging systems

- Half Synch/Half Async

  interconnect synchronous and asynchronous components

Composition Patterns

- Scatter/Gather

  send the same message to multiple recipients which will/may reply

- Canary Call

  avoid crashing all recipients of a poisoned request

- Master/Slave

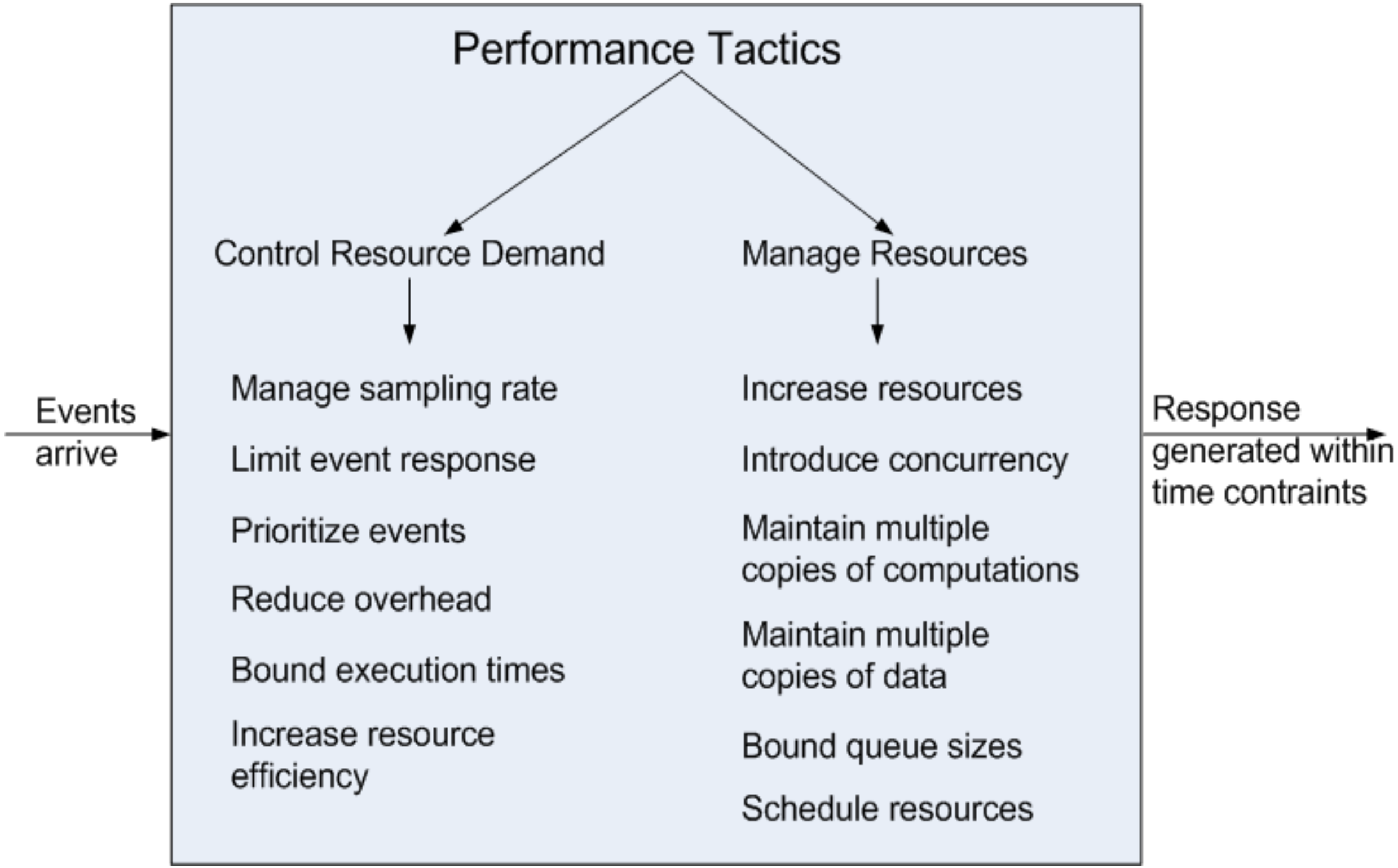
  speed up the execution of long running computations

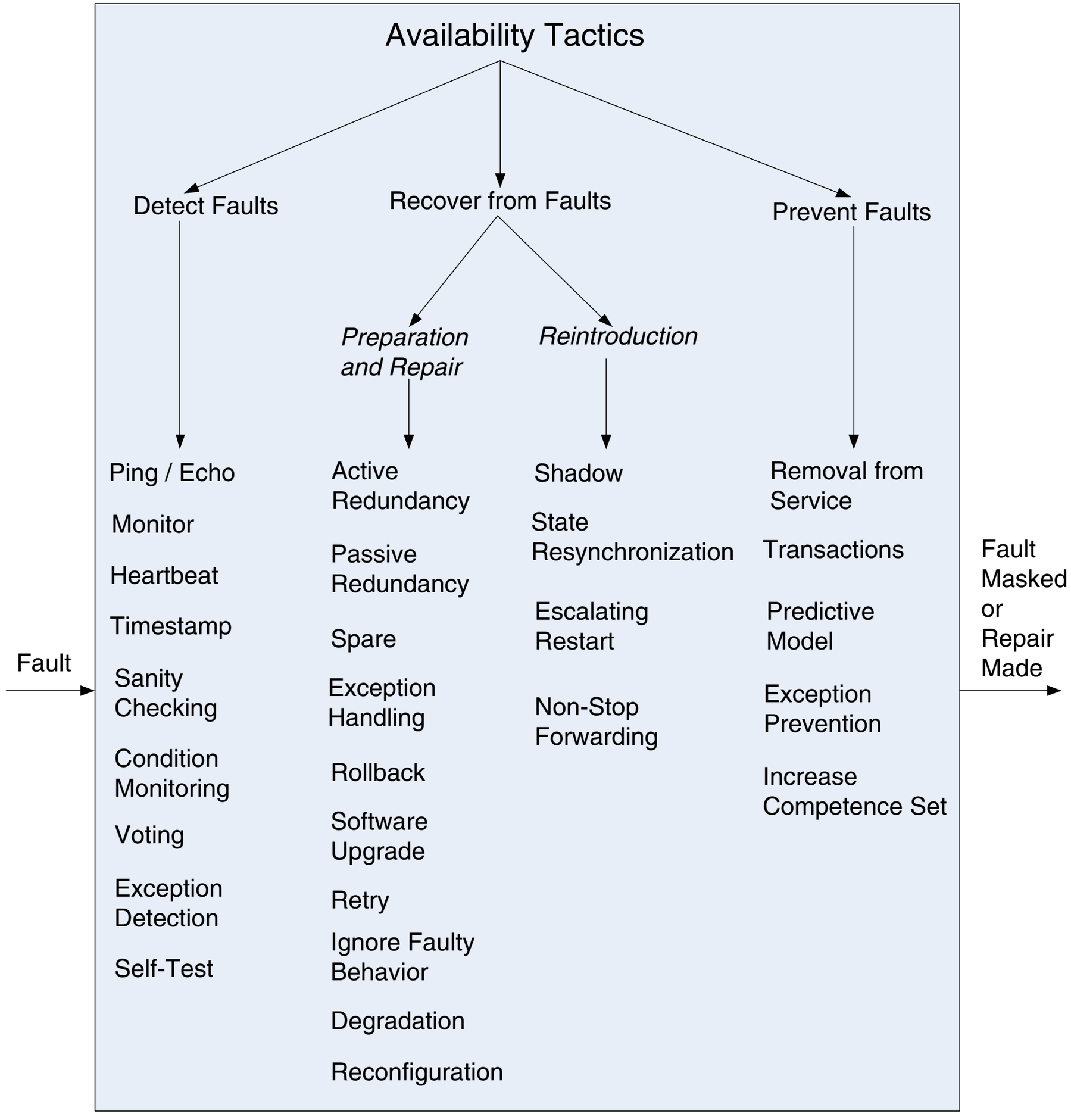
- Load Balancing

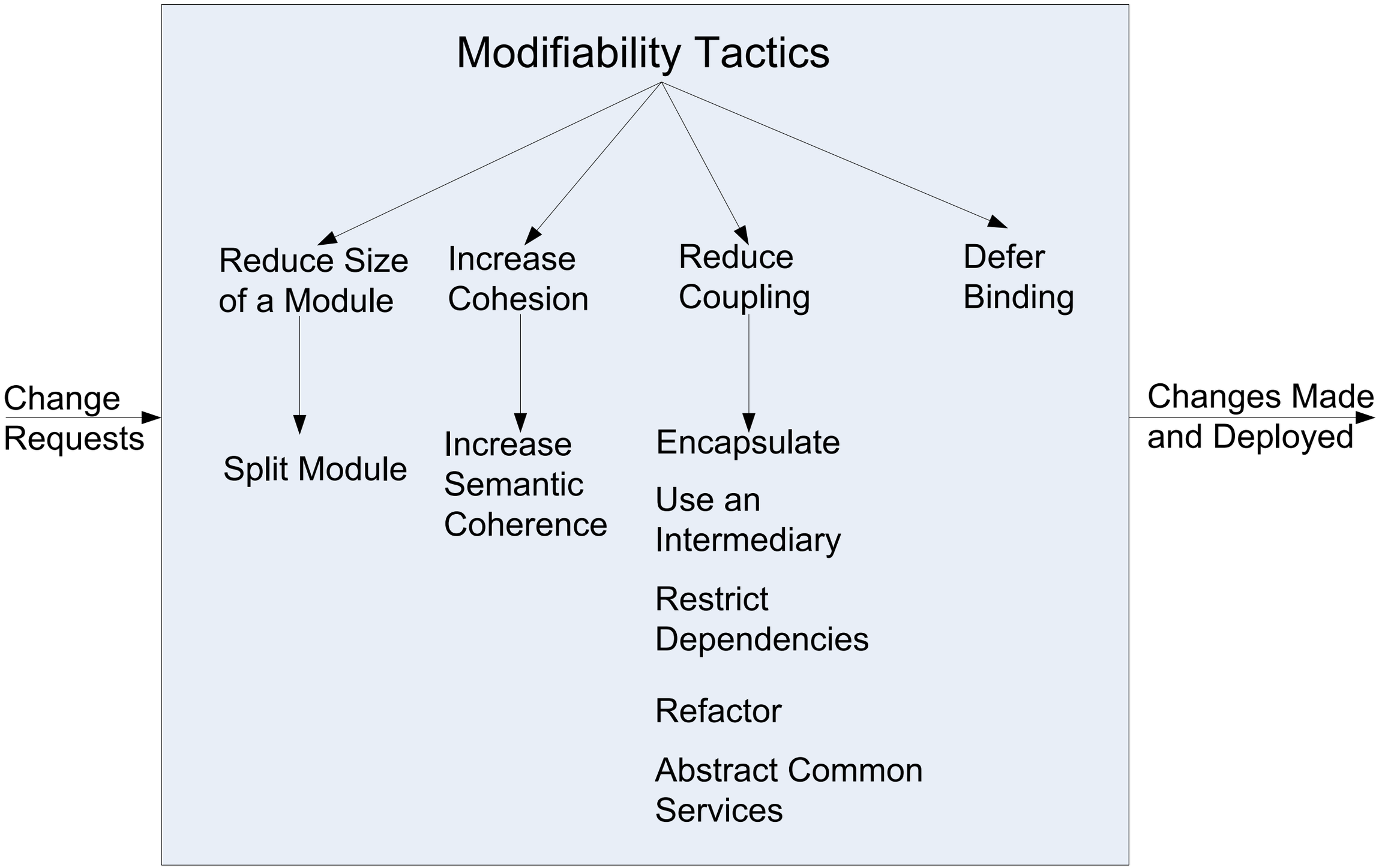
  speed up and scale up the execution of requests of many clients

- Orchestration

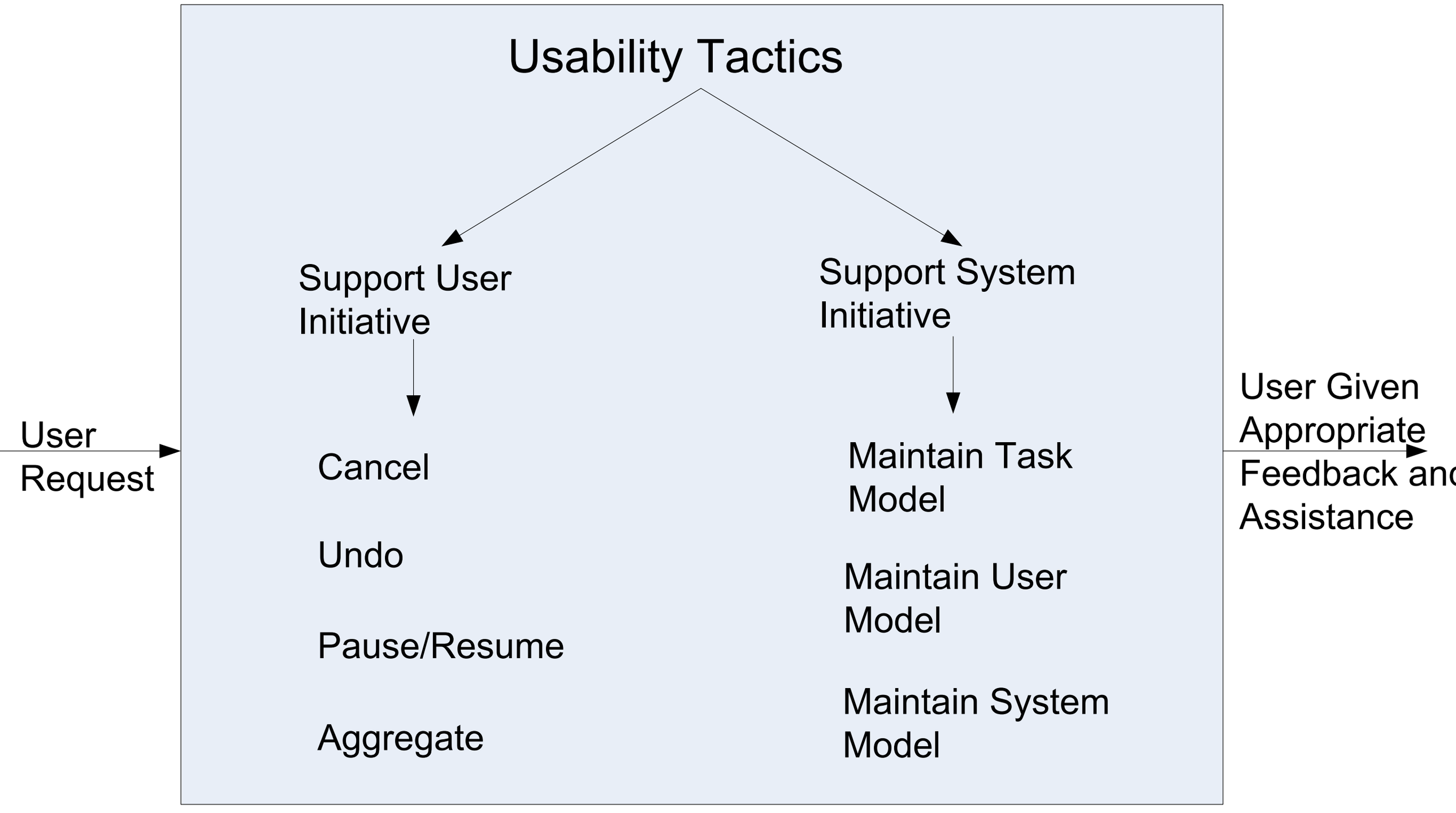
  improve the reuse of existing applications


Performance Tactics
Control Resource Demand
Manage Resources
Manage sampling rate
Increase resources
Limit event response
Introduce concurrency
Prioritize events
Maintain multiple copies of computations
Reduce overhead
Maintain multiple copies of data
Bound execution times
Bound queue sizes
Increase resource efficiency
Schedule resources
Events arrive
Response generated within time contraints


Availability Tactics
Detect Faults
Recover from Faults
Prevent Faults
Preparation and Repair
Reintroduction
Ping / Echo
Monitor
Heartbeat
Timestamp
Sanity Checking
Condition Monitoring
Voting
Exception Detection
Self-Test
Active Redundancy
Passive Redundancy
Spare
Exception Handling
Rollback
Software Upgrade
Retry
Ignore Faulty Behavior
Degradation
Reconfiguration
Shadow
State Resynchronization
Escalating Restart
Non-Stop Forwarding
Removal from Service
Transactions
Predictive Model
Exception Prevention
Increase Competence Set
Fault
Fault Masked or Repair Made


Modifiability Tactics
Reduce Size of a Module
Increase Cohesion
Reduce Coupling
Defer Binding
Split Module
Increase Semantic Coherence
Encapsulate
Use an Intermediary
Restrict Dependencies
Refactor
Abstract Common Services
Change Requests
Changes Made and Deployed


Testability Tactics
Control and Observe System State
Limit Complexity
Tests Executed
Specialized Interfaces
Limit Structural Complexity
Faults Detected
Record/ Playback
Limit Non-determinism
Localize State Storage
Abstract Data Sources
Sandbox
Executable Assertions


Usability Tactics
Support User Initiative
Support System Initiative
User Request
Cancel
Undo
Pause/Resume
Aggregate
Maintain Task Model
Maintain User Model
Maintain System Model
User Given Appropriate Feedback and Assistance