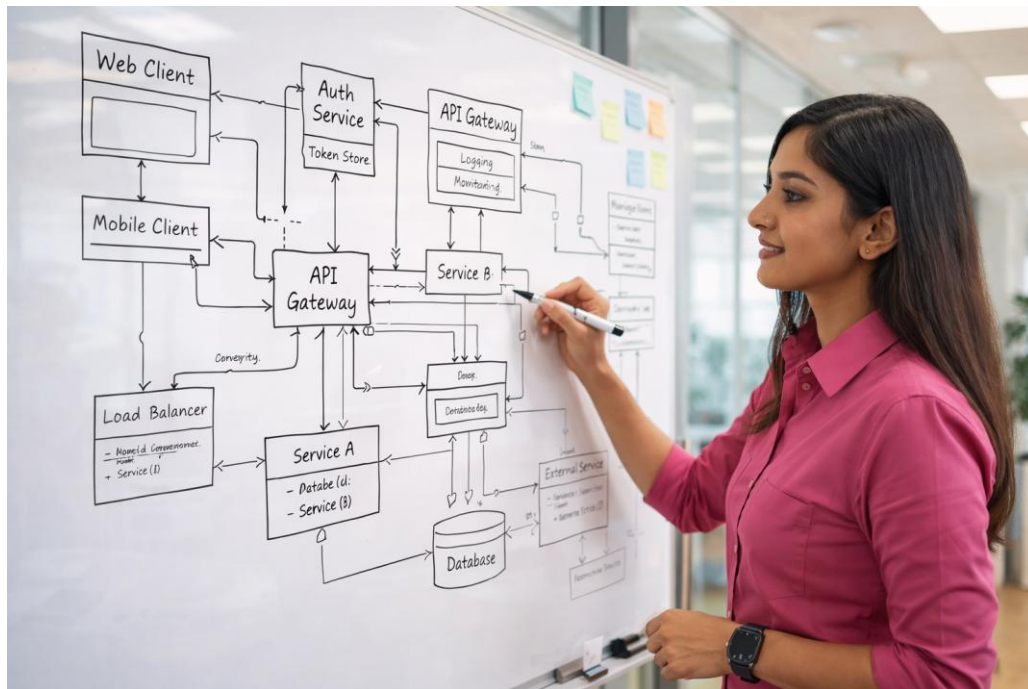


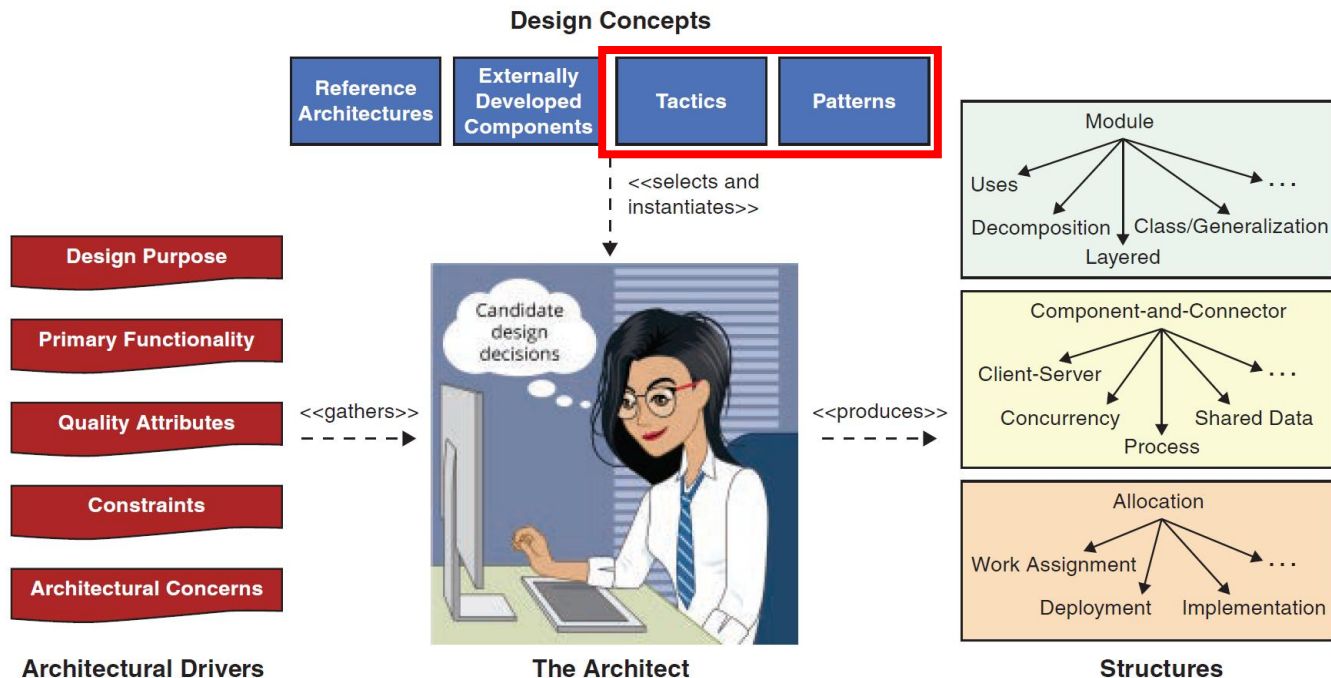
CS3213: Foundations of Software Engineering

Architectural Tactics and Patterns

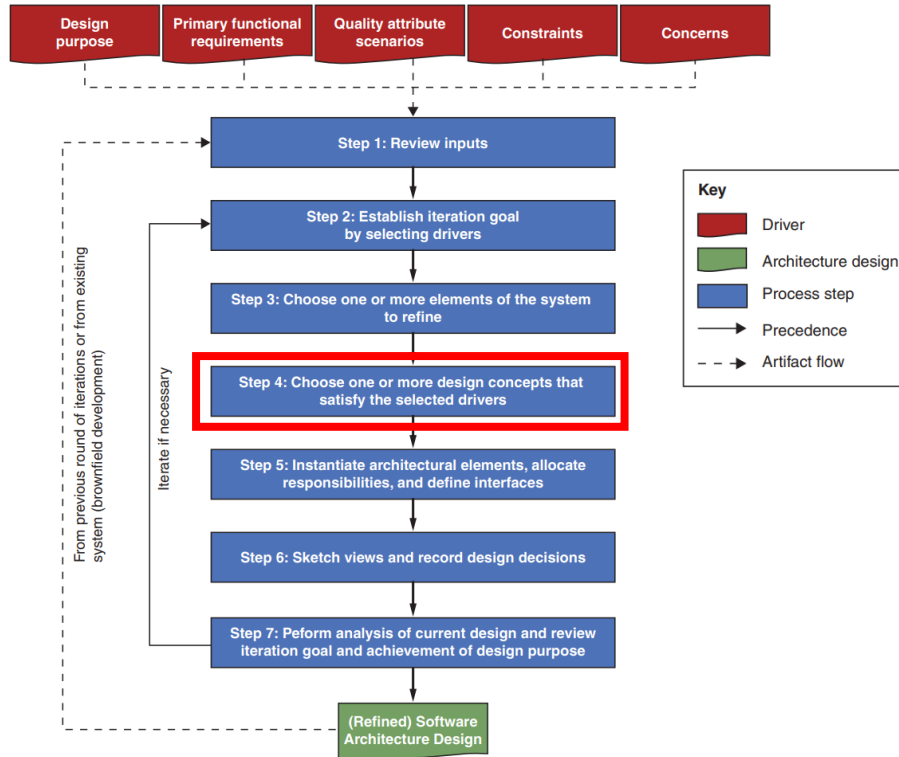
Architectural Tactics and Patterns



Approaching Architecture

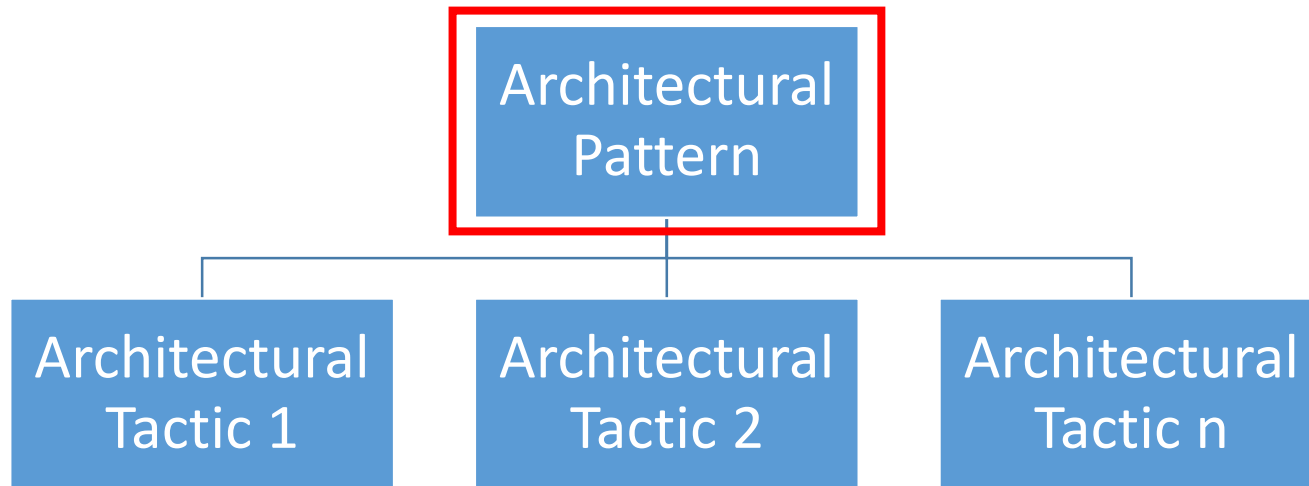


Attribute-Driven Design (ADD)



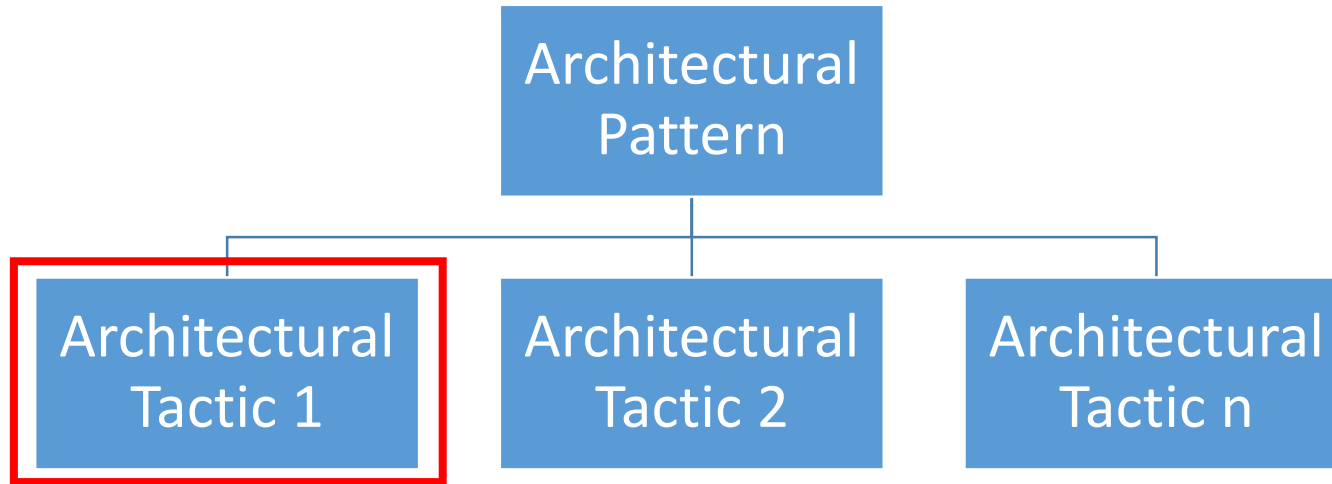
Architectural Tactics and Patterns

typically comprise multiple architectural tactics



Architectural Tactics and Patterns

assign decision that influences a quality attribute



Quality Attributes

External quality	Brief description
Availability	The extent to which the system's services are available when and where they are needed
Installability	How easy it is to correctly install, uninstall, and reinstall the application
Integrity	The extent to which the system protects against data inaccuracy and loss
Interoperability	How easily the system can interconnect and exchange data with other systems or components
Performance	How quickly and predictably the system responds to user inputs or other events
Reliability	How long the system runs before experiencing a failure
Robustness	How well the system responds to unexpected operating conditions
Safety	How well the system protects against injury or damage
Security	How well the system protects against unauthorized access to the application and its data
Usability	How easy it is for people to learn, remember, and use the system
Internal quality	Brief description
Efficiency	How efficiently the system uses computer resources
Modifiability	How easy it is to maintain, change, enhance, and restructure the system
Portability	How easily the system can be made to work in other operating environments
Reusability	To what extent components can be used in other systems
Scalability	How easily the system can grow to handle more users, transactions, servers, or other extensions
Verifiability	How readily developers and testers can confirm that the software was implemented correctly

Quality Attributes

2024 IEEE/ACM 46th International Conference on Software Engineering: Software Engineering in Society (ICSE-SEIS)

A Synthesis of Green Architectural Tactics for ML-Enabled Systems

Heli Järvenpää
Vrije Universiteit Amsterdam
Amsterdam, The Netherlands
h.m.jarvenpaa@student.vu.nl

Patricia Lago
Vrije Universiteit Amsterdam
Amsterdam, The Netherlands
p.lago@vu.nl

Justus Bogner
Vrije Universiteit Amsterdam
Amsterdam, The Netherlands
j.bogner@vu.nl

Grace Lewis
Carnegie Mellon Software
Engineering Institute
Pittsburgh, PA, USA
glewis@sei.cmu.edu

Henry Muccini
FrAmELab, University of L'Aquila
L'Aquila, Italy
henry.muccini@univaq.it

Ipek Ozkaya
Carnegie Mellon Software
Engineering Institute
Pittsburgh, PA, USA
ozkaya@sei.cmu.edu

ABSTRACT

The rapid adoption of artificial intelligence (AI) and machine learning (ML) has generated growing interest in understanding their environmental impact and the challenges associated with designing environmentally friendly ML-enabled systems. While Green AI research, i.e., research that tries to minimize the energy footprint of AI, is receiving increasing attention, very few concrete guidelines are available on how ML-enabled systems can be designed to be more environmentally sustainable. In this paper, we provide a catalog of 30 green architectural tactics for ML-enabled systems to fill this gap. An architectural tactic is a high-level design technique to improve software quality, in our case environmental sustainability. We derived the tactics from the analysis of 51 peer-reviewed publications that primarily explore Green AI, and validated them using a focus group approach with three experts. The 30 tactics we identified are aimed to serve as an initial reference guide for further exploration into Green AI from a software engineering perspective, and assist in designing sustainable ML-enabled systems. To enhance transparency and facilitate their widespread use and extension, we make the tactics available online in easily consumable formats. Wide-spread adoption of these tactics has the potential to substantially reduce the societal impact of ML-enabled systems regarding their energy and carbon footprint.

CCS CONCEPTS

• Software and its engineering → Designing software; Software architectures; • Social and professional topics → Sustainability; • Computing methodologies → Machine learning,

ACM Reference Format:

Heli Järvenpää, Patricia Lago, Justus Bogner, Grace Lewis, Henry Muccini, and Ipek Ozkaya. 2024. A Synthesis of Green Architectural Tactics for ML-Enabled Systems. In *Software Engineering in Society (ICSE-SEIS '24)*, April 14–20, 2024, Lisbon, Portugal. ACM, New York, NY, USA, 12 pages. <https://doi.org/10.1145/3639475.3640111>

Lay Abstract: Machine learning (ML) is a technology field that wants to provide software with functionality similar to human-like intelligence, e.g., for understanding text or describing images. However, creating and using systems with ML needs a lot more computing power than non-ML systems, which is bad for the environment. Companies therefore need concrete advice on how they can create ML systems that are environmentally sustainable. In this paper, we provide a catalog of 30 green architectural tactics for these systems. An architectural tactic is a high-level design technique to improve software quality, in our case environmental sustainability. To achieve this, we analyzed 51 scientific papers and later discussed with 3 experts to improve and extend our catalog. If many companies start using these tactics, the energy footprint of systems with ML can be greatly reduced.

1 INTRODUCTION

Artificial intelligence (AI) and machine learning (ML) have shown significant potential in digital innovations, with a growing number of different ML applications expanding across a wide spectrum of industries, from healthcare to agriculture and management [41]. This rapid growth of ML applications has also drawn attention to its

Table 4: Green Tactics Related to Model Training

Tactic	Description	Target QA	Source
T18: Use quantization-aware training	Convert high-precision data types to lower precision during training	Accuracy*	[26, 50]
T19: Use checkpoints during training	Use checkpoints to avoid a knowledge loss in case of a premature termination	Recoverability*	[48]
T20: Design for memory constraints	Consider possible memory constraints during training	Recoverability*	[48]

The * means energy efficiency was considered as a secondary QA



Architectural Tactics

Examples: Quality Attributes

External quality	Brief description
Availability	The extent to which the system's services are available when and where they are needed
Installability	How easy it is to correctly install, uninstall, and reinstall the application
Integrity	The extent to which the system protects against data inaccuracy and loss
Interoperability	How easily the system can interconnect and exchange data with other systems or components
Performance	How quickly and predictably the system responds to user inputs or other events
Reliability	How long the system runs before experiencing a failure
Robustness	How well the system responds to unexpected operating conditions
Safety	How well the system protects against injury or damage
Security	How well the system protects against unauthorized access to the application and its data
Usability	How easy it is for people to learn, remember, and use the system
Internal quality	Brief description
Efficiency	How efficiently the system uses computer resources
Modifiability	How easy it is to maintain, change, enhance, and restructure the system
Portability	How easily the system can be made to work in other operating environments
Reusability	To what extent components can be used in other systems
Scalability	How easily the system can grow to handle more users, transactions, servers, or other extensions
Verifiability	How readily developers and testers can confirm that the software was implemented correctly



Terminology

- **Mistake:** human act or decision resulting in an error
- **Defect (bug):** an error in the program code
- **Fault:** the location in which the programming error is triggered and we enter an unwanted state
- **Failure:** where the fault becomes externally visible

Terminology

Mistake: human act or decision resulting in an error



Terminology

Defect (bug): error in the program code



Defect (Bug)

Terminology

Fault: the location in which the programming error is triggered and we enter an unwanted state



Defect (Bug)

Fault

Terminology

Failure: where the fault becomes externally visible



Defect (Bug)

Fault

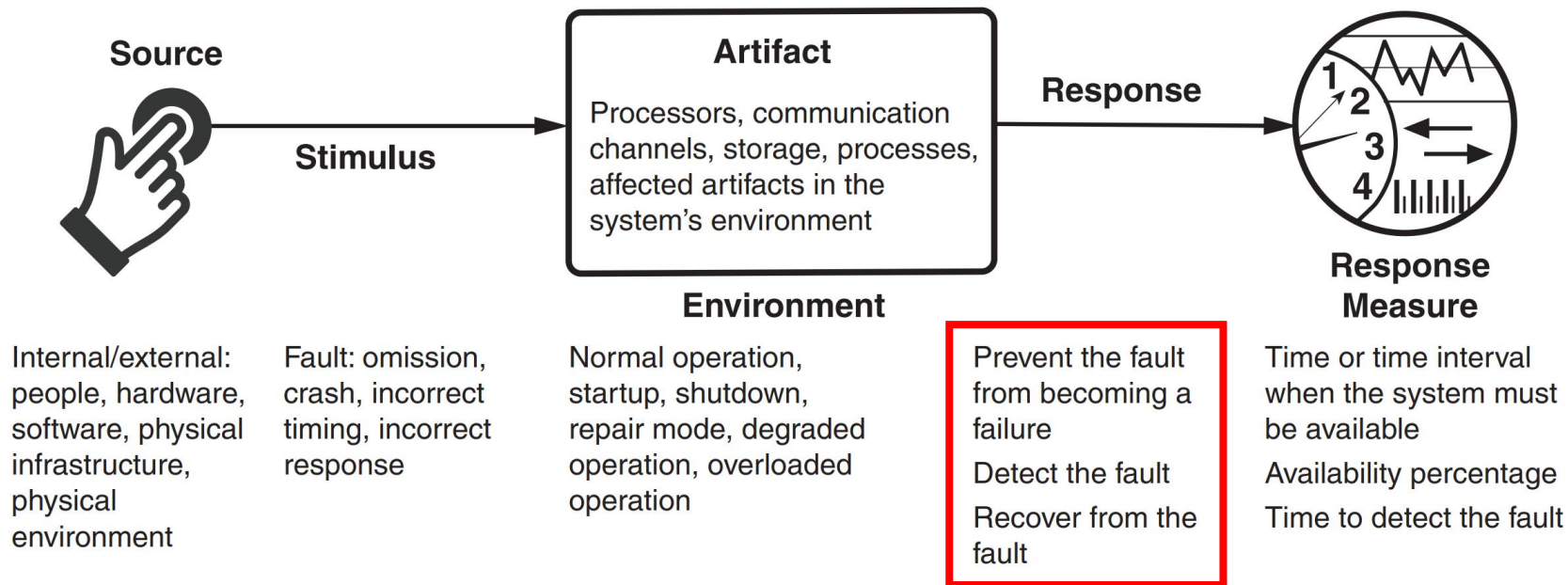
Failure



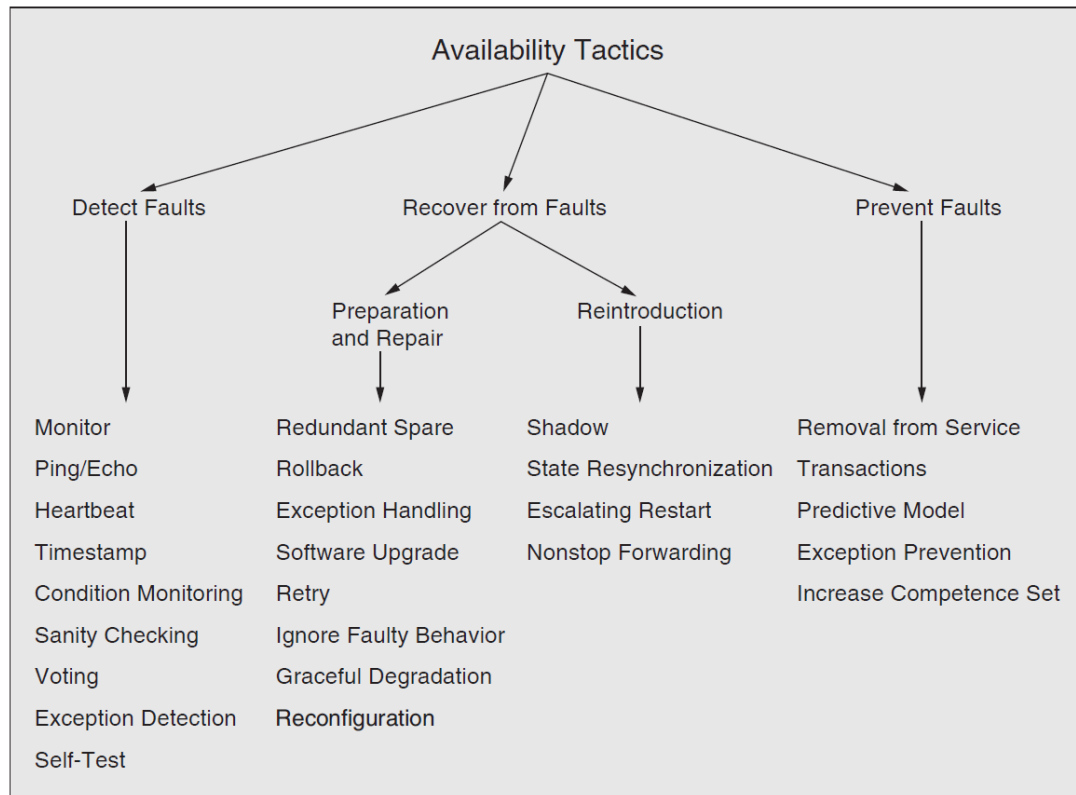
Example: Availability

- Availability: software can operate when you need it to be
- Related to reliability, but includes the notion of recoverability
 - The system can repair itself or mask faults

Quality Attribute Scenarios: Availability

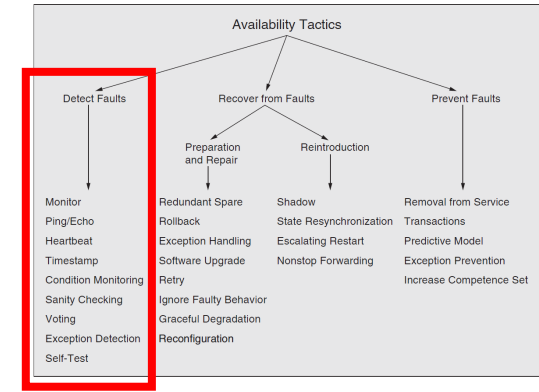


Availability Tactics



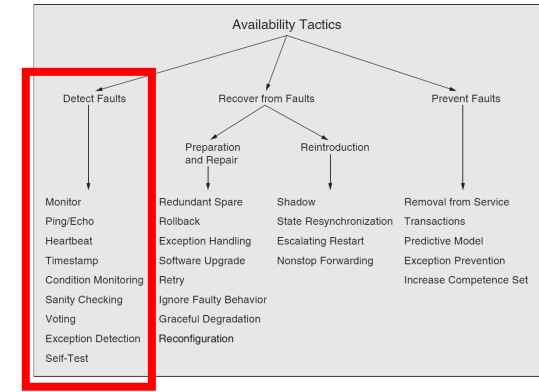
Detect Faults

- **Monitor:** monitor various components of the system (e.g., using a system monitor)



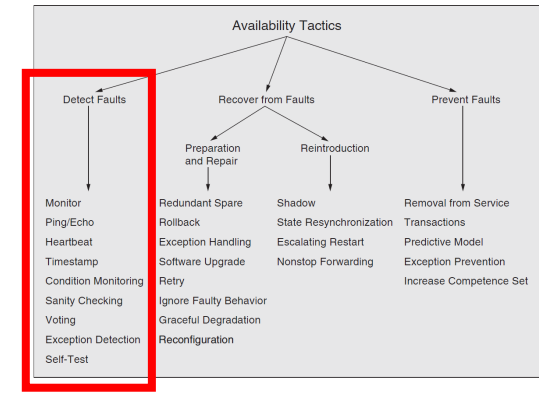
Detect Faults

- **Monitor:** monitor various components of the system (e.g., using a system monitor)
- **Heartbeat:** periodic message exchange between a system monitor and a process being monitored

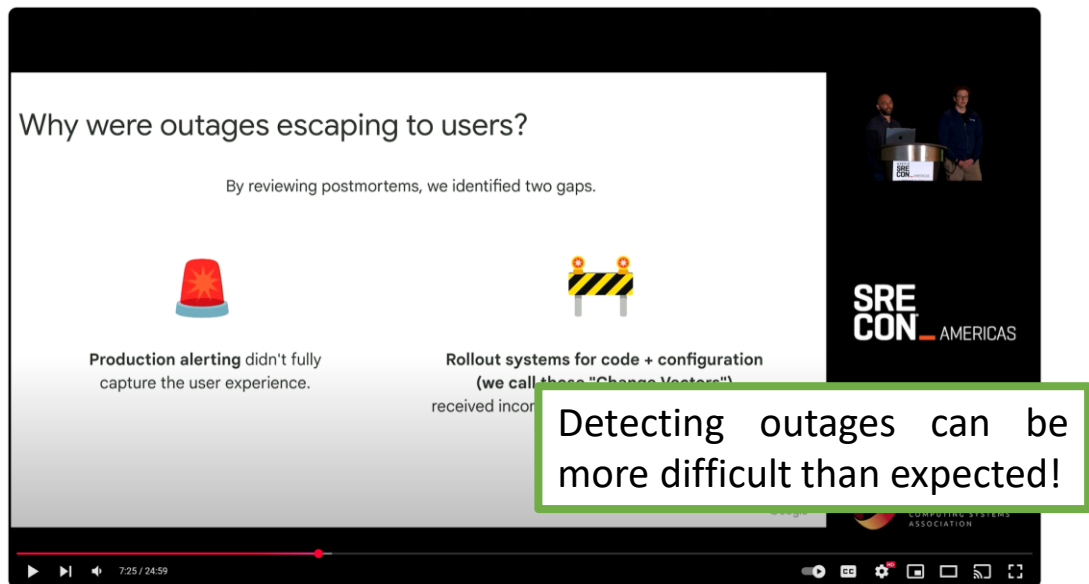


Detect Faults

- **Monitor:** monitor various components of the system (e.g., using a system monitor)
- **Heartbeat:** periodic message exchange between a system monitor and a process being monitored
- **Voting:** compare the results of multiple sources that should produce the same result



Example: Monitoring at Google Maps



SREcon24 Americas - Product Reliability for Google Maps

SRECon

Example: Faulty CPUs by Google

To efficiently screen the fleet, SiliFuzz needs to spend a non-trivial amount of time on every core of every machine. Machines in Google's data centers **are continuously screened for SDC defects** with a collection of tests, including SiliFuzz. This is done while machines are in use in production. To minimize the performance impact of this screening, only a small number of machines in a cluster are being tested at any given time.

SiliFuzz: Fuzzing CPUs by proxy

Kostya Serebryany
Google
kcc@google.com

Maxim Lifantsev
Google
maxim@google.com

Konstantin Shtoyk
Google
kostik@google.com

Doug Kwan
Google
dougkwan@google.com

Peter Hochschild
Google
phoch@google.com

Abstract

CPUs are becoming more complex with every generation, at both the logical and the physical levels. This potentially leads to more logic bugs and electrical defects in CPUs being overlooked during testing, which causes data corruption or other undesirable effects when these CPUs are used in production. These ever-present problems may also have simply become more evident as more CPUs are operated and monitored by large cloud providers.

If the RTL ("source code") of a CPU were available, we could apply *greybox fuzzing* to the CPU model almost as we do to any other software [1]. However our targets are general purpose x86_64 CPUs produced by third parties, where we do not have the RTL design, so in our case CPU implementations are opaque. Moreover, we are more interested in electrical *defects* as opposed to logic

1 Introduction

When discussing misbehaving CPUs we distinguish between *logic bugs* and *electrical defects*. A **logic bug** is an invalid CPU behavior inherent to a particular CPU microarchitecture or stepping, e.g. [2, 3, 4]. An **electrical defect** is an invalid CPU behavior that happens only on one or several chips. Often, but not always, an electrical defect affects only one of the physical cores on the chip. A defect may be present in a CPU from day one (if it was missed by the vendor during testing) or it may be a result of physical wear-out (e.g., circuit aging). For the purposes of this paper the distinction between day-one defects and wear-out is not important.

The term "**Silent Data Corruption**" (SDC) has been coined [5, 6, 7, 8] to represent CPU defects (and bugs) that do not cause an immediate observable failure, but instead silently lead to incorrect computation. This phenomenon has been known for

Example: Availability and Radiation

Compiling an application for use in highly radioactive environments [closed]

Ask Question

Asked 9 years, 9 months ago Modified 1 year, 2 months ago Viewed 727k times



1639



Closed. This question needs to be more [focused](#). It is not currently accepting answers.

Want to improve this question? Guide the asker to update the question so it focuses on a single, specific problem. Narrowing the question will help others answer the question concisely. You may [edit the question](#) if you feel you can improve it yourself. If edited, the question will be reviewed and might be reopened.

Closed 12 months ago.

Improve this question

We are compiling an embedded C++ application that is deployed in a shielded device in an environment bombarded with [ionizing radiation](#). We are using GCC and cross-compiling for ARM. When deployed, our application generates some erroneous data and crashes more often than we would like. The hardware is designed for this environment, and our application has run on this platform for several years.

Are there changes we can make to our code, or compile-time improvements that can be made to identify/correct [soft errors](#) and memory-corruption caused by [single event upsets](#)? Have any other developers had success in reducing the harmful effects of soft errors on a long-running application?

c++ c gcc embedded fault-tolerance

The Overflow Blog

- How Stack Overflow is taking on spam and bad actors
- How AWS re:Invented the cloud

Featured on Meta

- Community Asks Sprint Announcement – January 2026: Custom site-specific badges!
- All users on Stack Overflow can now participate in chat
- Should Stack Overflow utilize machine learning anti-spam?
- Policy: Generative AI (e.g., ChatGPT) is banned

AD

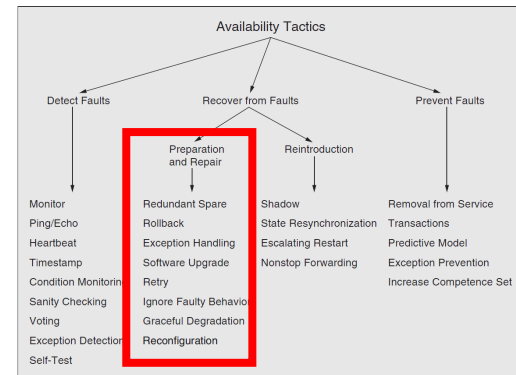
Community activity

Last 1 hr

- 5193 users online
- 7 questions
- 10 answers
- 36 comments

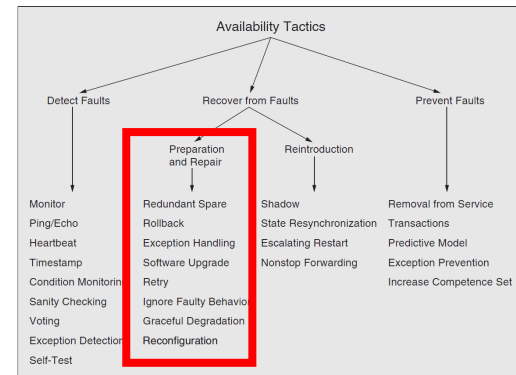
Recover: Preparation and Repair

- **Redundant spare:** one or more duplicate components can step in if a component fails



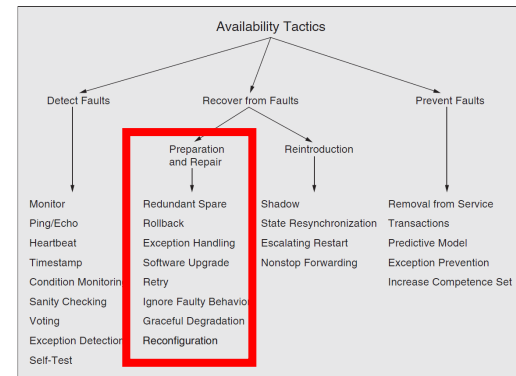
Recover: Preparation and Repair

- **Redundant spare:** one or more duplicate components can step in if a component fails
- **Rollback:** revert to a previous, known good state



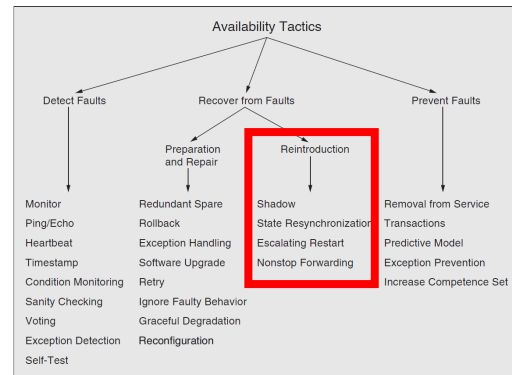
Recover: Preparation and Repair

- **Redundant spare:** one or more duplicate components can step in if a component fails
- **Rollback:** revert to a previous, known good state
- **Graceful degradation:** maintain most critical system functions



Recover: Reintroduction

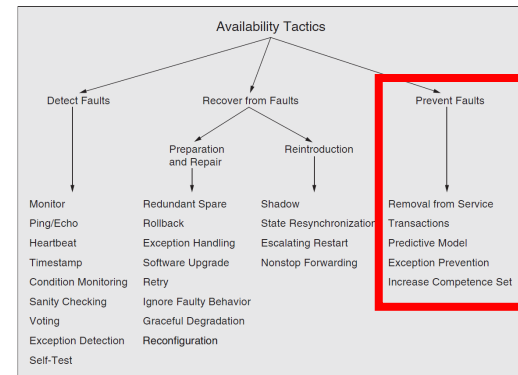
- **Shadow:** operate a component in a “shadow mode” while being monitored before reverting it back to an active mode
- **Escalating restart:** Automatic restart at different granularities (e.g., lowest level might clear caches, while highest one restarts the whole system)



Prevent Faults

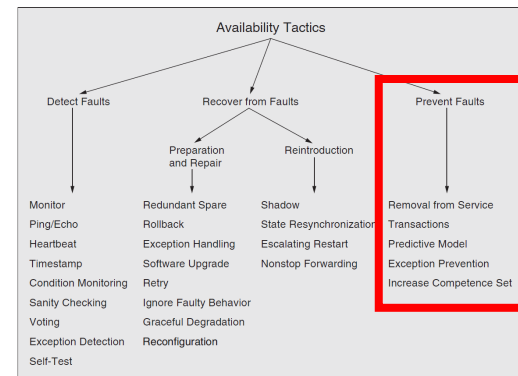
- **Removal from service:** Temporarily placing a system component in an out-of-service state to mitigate potential faults (e.g., suspected memory leak)

Machines that are believed to be defective **are taken out of production and set aside for detailed diagnostics**. These machines are put into a quarantine pool so that we can run a number of tests on them. Testing usually takes weeks for a machine since some failures are dependent on frequency, voltage and temperature and we need to vary these parameters to run our tests with different combinations.



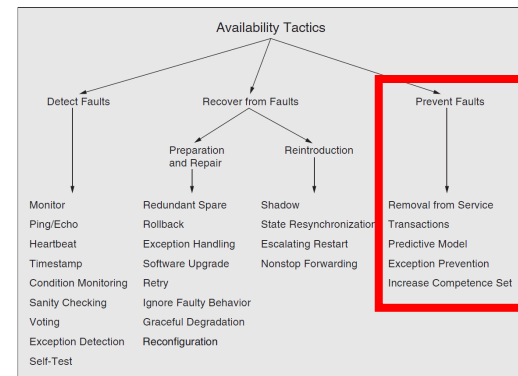
Prevent Faults

- **Removal from service:** Temporarily placing a system component in an out-of-service state to mitigate potential faults (e.g., suspected memory leak)
- **Transactions:** Provide ACID properties (atomic, consistent, isolated, and durable)



Prevent Faults

- **Removal from service:** Temporarily placing a system component in an out-of-service state to mitigate potential faults (e.g., suspected memory leak)
- **Transactions:** Provide ACID properties (atomic, consistent, isolated, and durable)
- **Increase competence set:** set of states in which the program can “competently” operate

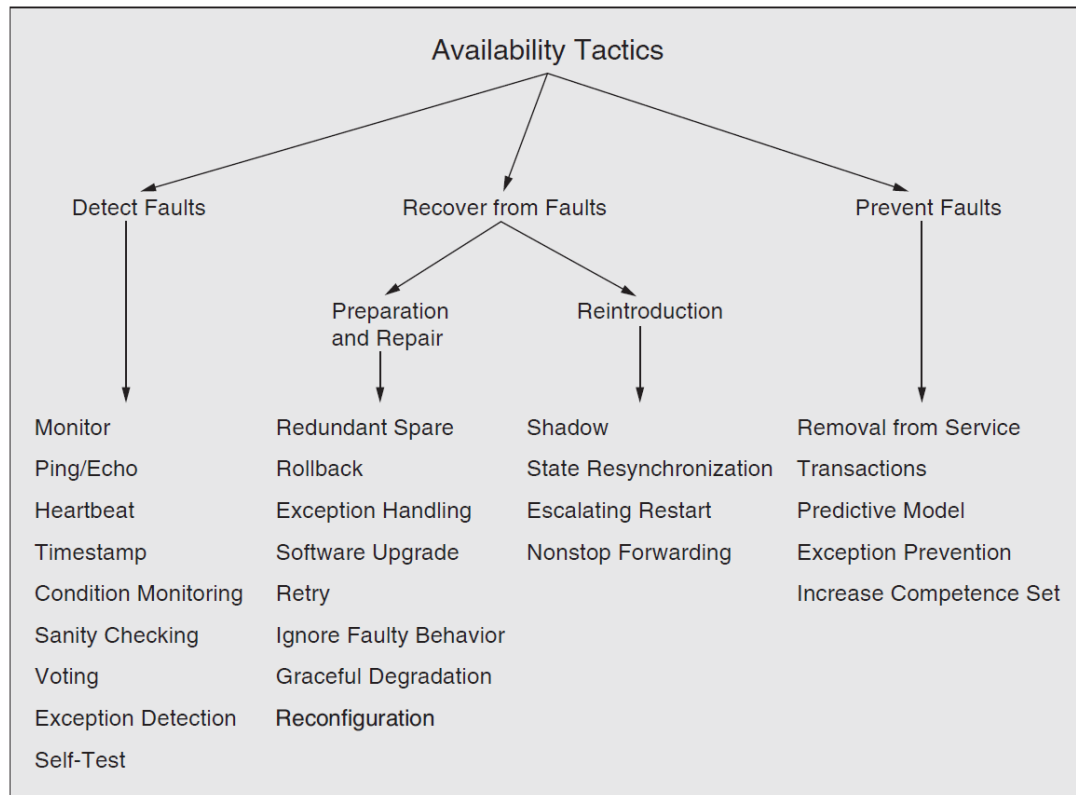


Increase Competence Set: SQLite

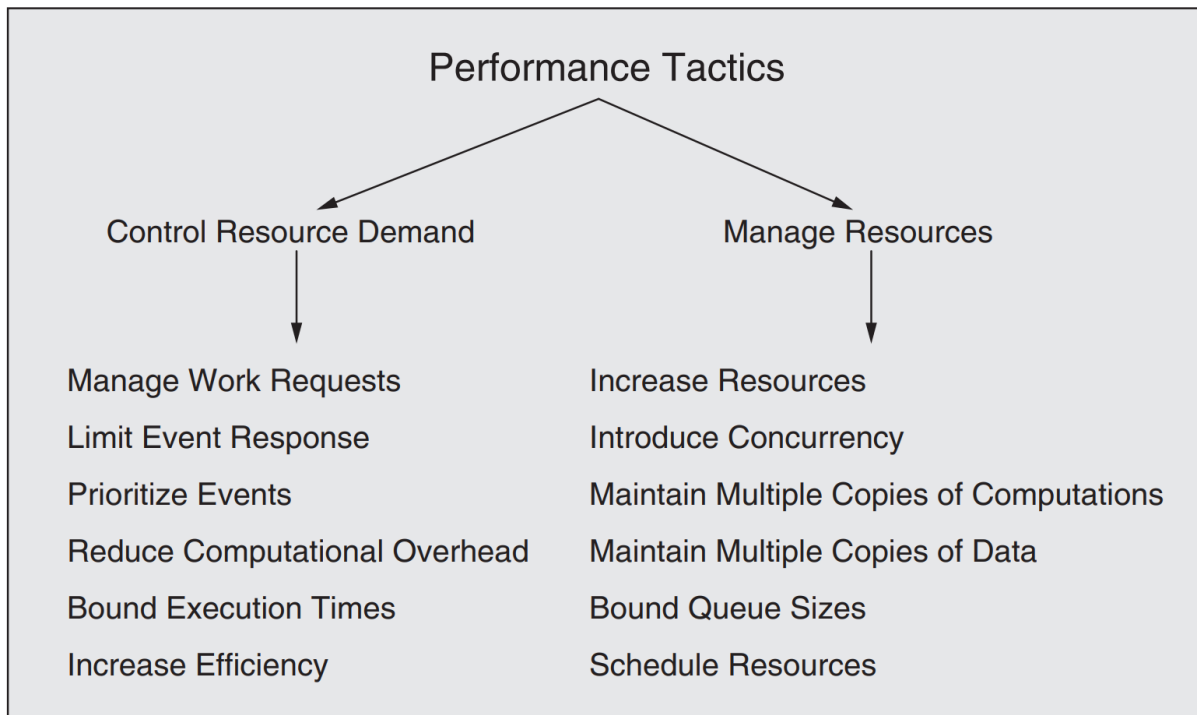
- SQLite is designed to operate correctly even in the presence of out-of-memory-error situations, I/O errors, crashes, ...

Anomaly tests are tests designed to verify the correct behavior of SQLite when something goes wrong. It is (relatively) easy to build an SQL database engine that behaves correctly on well-formed inputs on a fully functional computer. It is more **difficult to build a system that responds sanely to invalid inputs and continues to function following system malfunctions**. The anomaly tests are designed to verify the latter behavior.

Availability Tactics

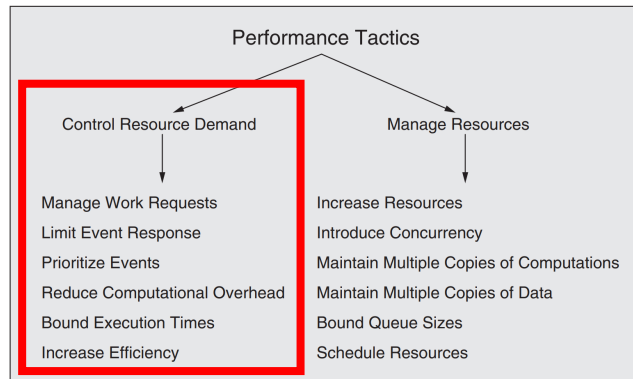


Performance Tactics



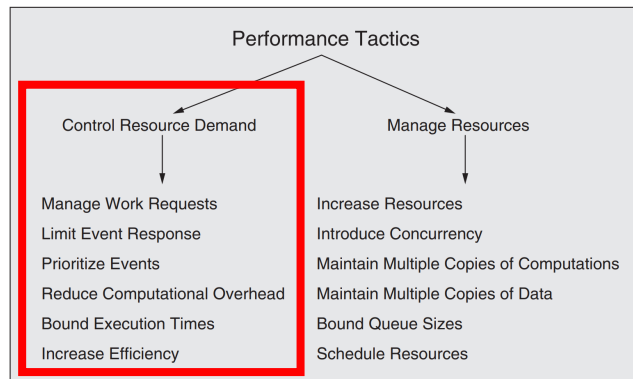
Control Resource Demand

- **Manage Work Requests:** reduces work by reducing the number of requests coming into the system or component



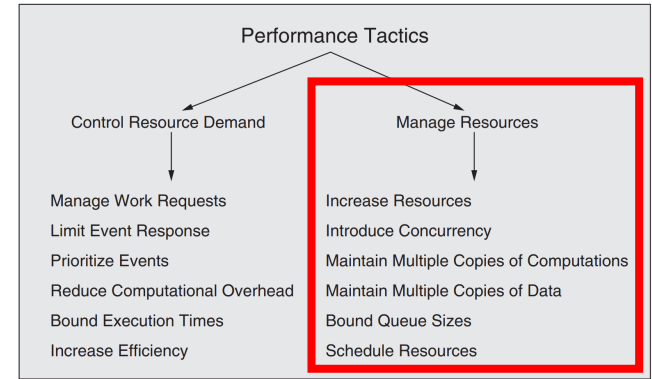
Control Resource Demand

- **Manage Work Requests:** reduces work by reducing the number of requests coming into the system or component
- **Limit Event Response:** policy for handling events too rapidly to be processed



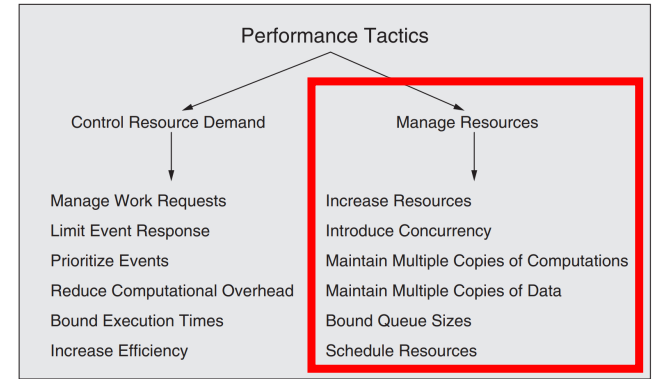
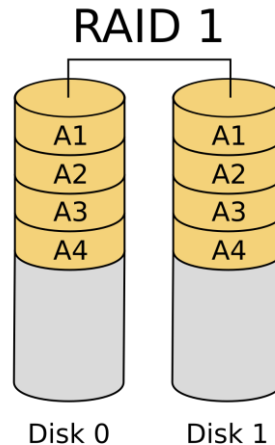
Manage Resources

- **Maintain multiple copies of computations:** involves keeping separate copies of the data to reduce the contention from multiple simultaneous accesses



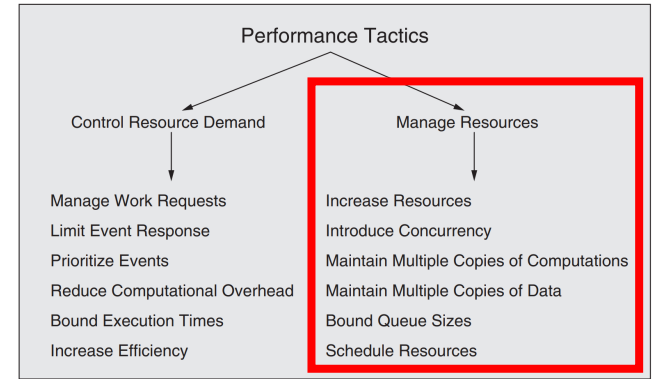
Manage Resources

- **Maintain multiple copies of computations:** involves keeping separate copies of the data to reduce the contention from multiple simultaneous accesses



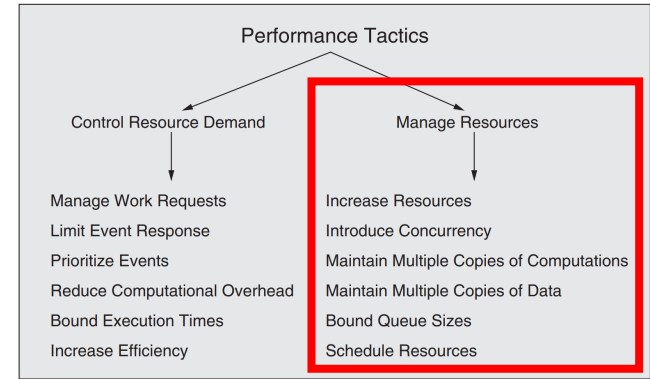
Manage Resources

- **Maintain multiple copies of computations:** involves keeping separate copies of the data to reduce the contention from multiple simultaneous accesses
- **Schedule Resources:** scheduling resources in the presence of contentions

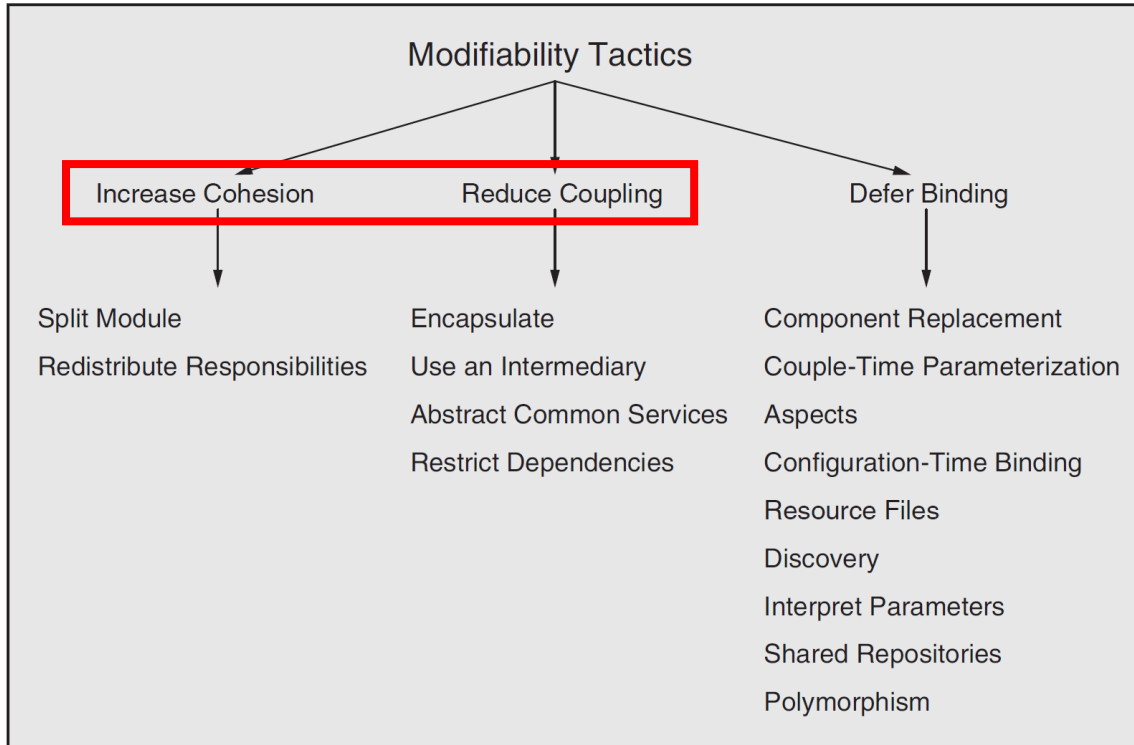


Manage Resources

- **Maintain multiple copies of computations:** involves keeping separate copies of the data to reduce the contention from multiple simultaneous accesses
- **Schedule Resources:** scheduling resources in the presence of contentions
 - First in, first out (FIFO)
 - Fixed-priority scheduling
 - Round-robin scheduling



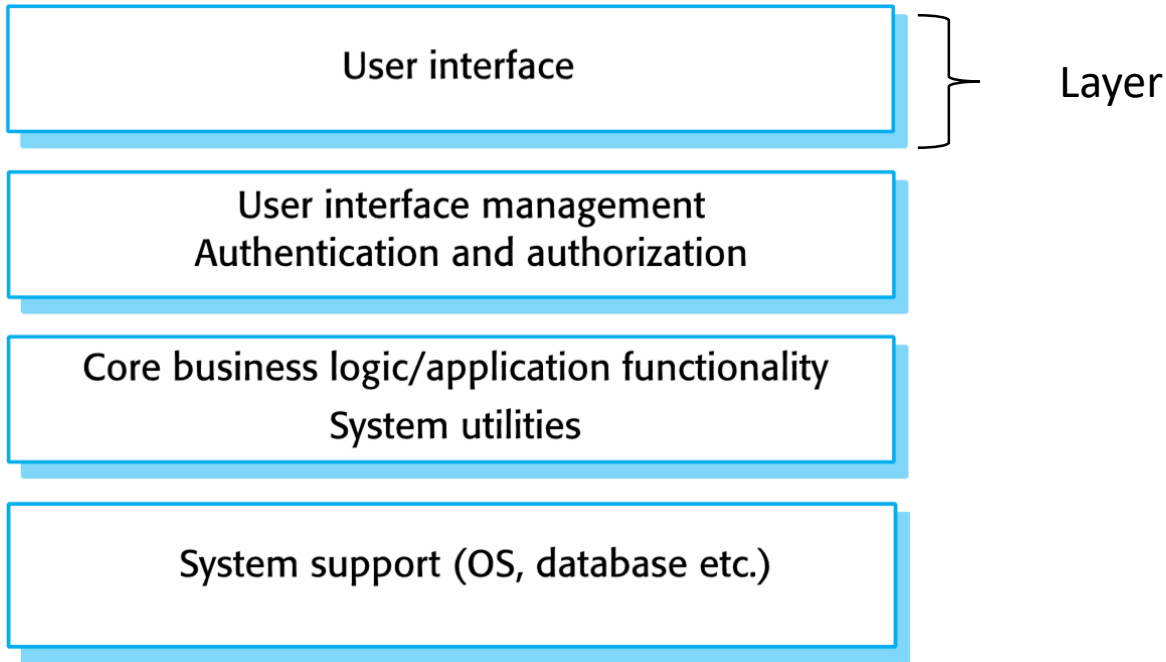
Modifiability Tactics





Architectural Patterns

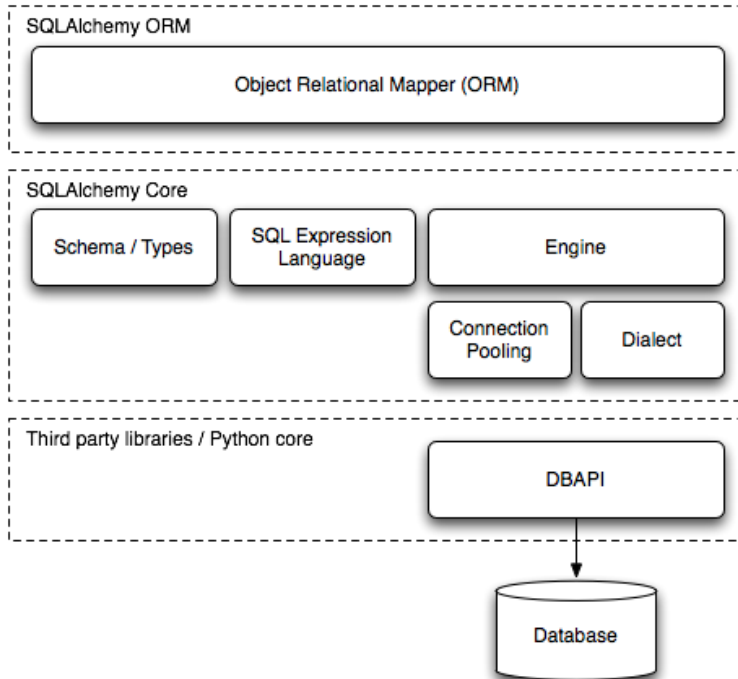
Layered Architecture



Kueh Lapis Architecture



SQLAlchemy





Layered Architecture: Benefits

- **Portability:** allows layers to be general or specific to an environment



Layered Architecture: Benefits

- **Portability:** allows layers to be general or specific to an environment
- **Reusability:** lower-level layers could be reused across different applications (e.g., consider a network communications layer in an OS)



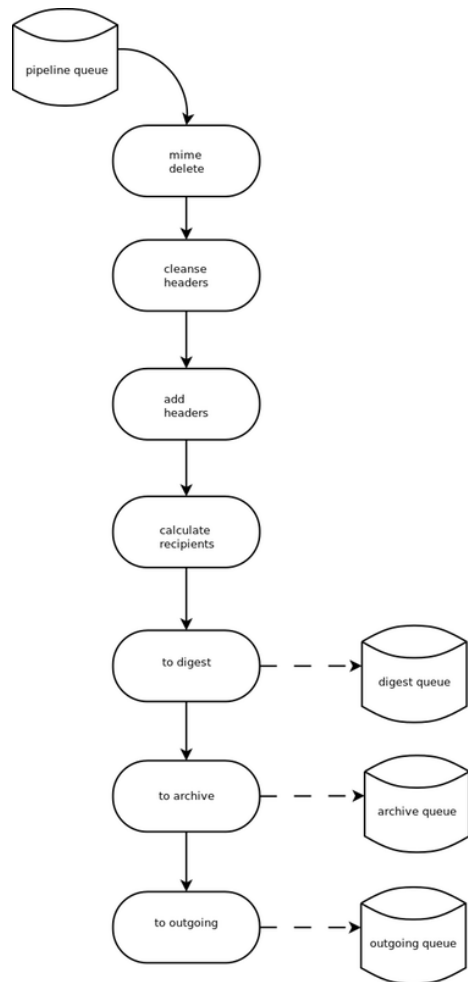
Layered Architecture: Benefits

- **Portability:** allows layers to be general or specific to an environment
- **Reusability:** lower-level layers could be reused across different applications (e.g., consider a network communications layer in an OS)
- **Modifiability:** lower layers can easily be changed without affecting upper layers

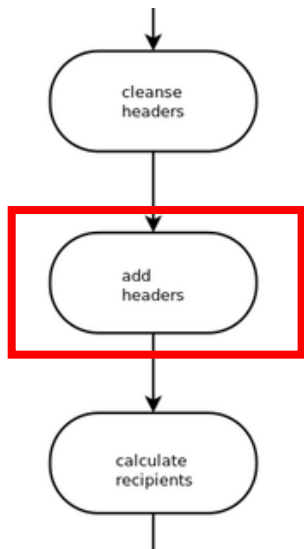
Pipe-and-Filter Architecture



Example: Mailman



Example: Mailman



Precedence: header



Example: Pipe-and-Filter

```
ps aux | grep root | wc -l
```



Pipe-and-Filter Architecture: Benefits

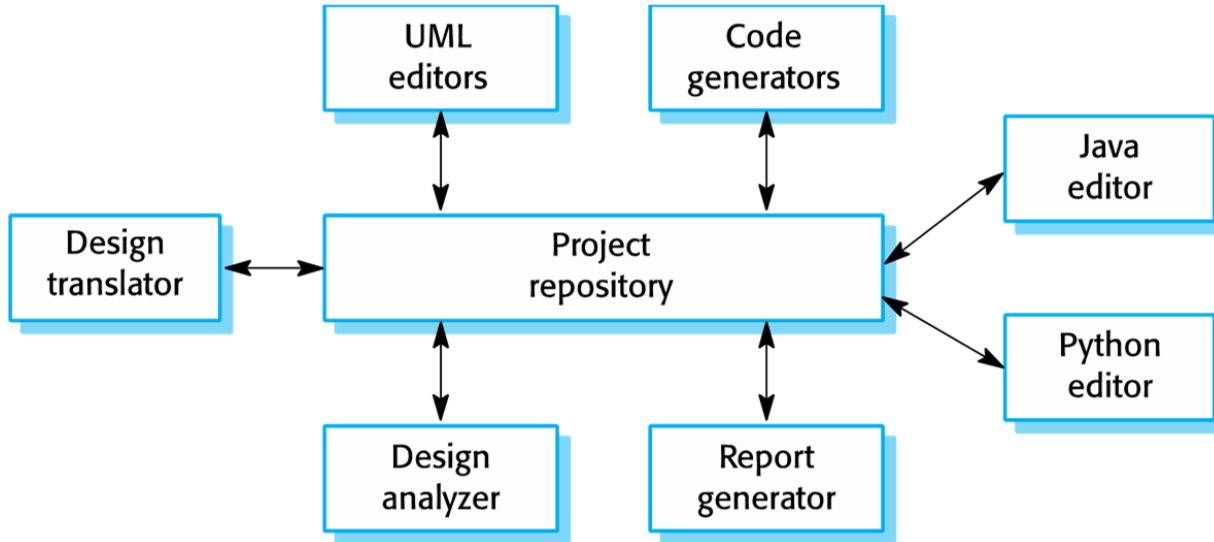
- **Modifiability:** filters are independent from each other and can be modified independently



Pipe-and-Filter Architecture: Benefits

- **Modifiability**: filters are independent from each other and can be modified independently
- **Reconfigurability**: filters can be combined in different ways

Model-centered Architecture



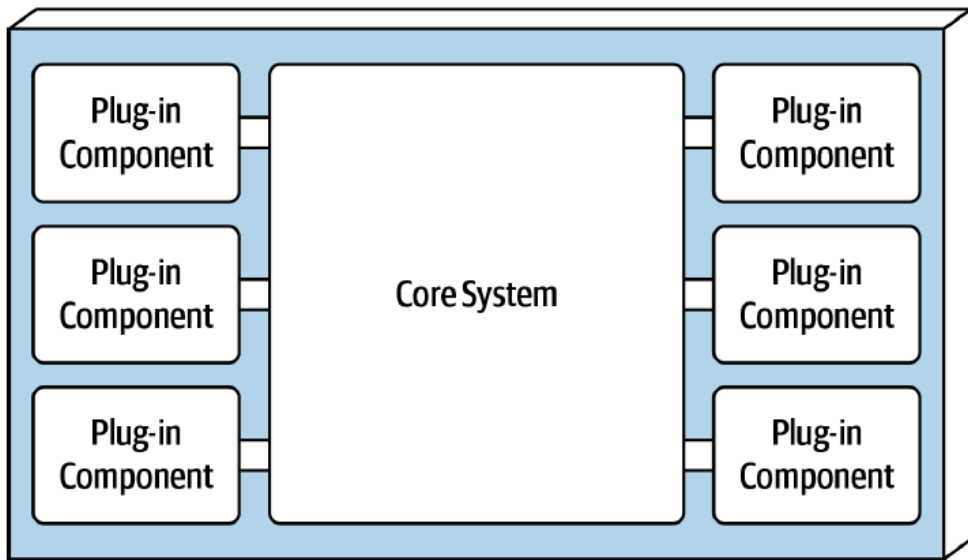


Model-centered Architecture: Benefits

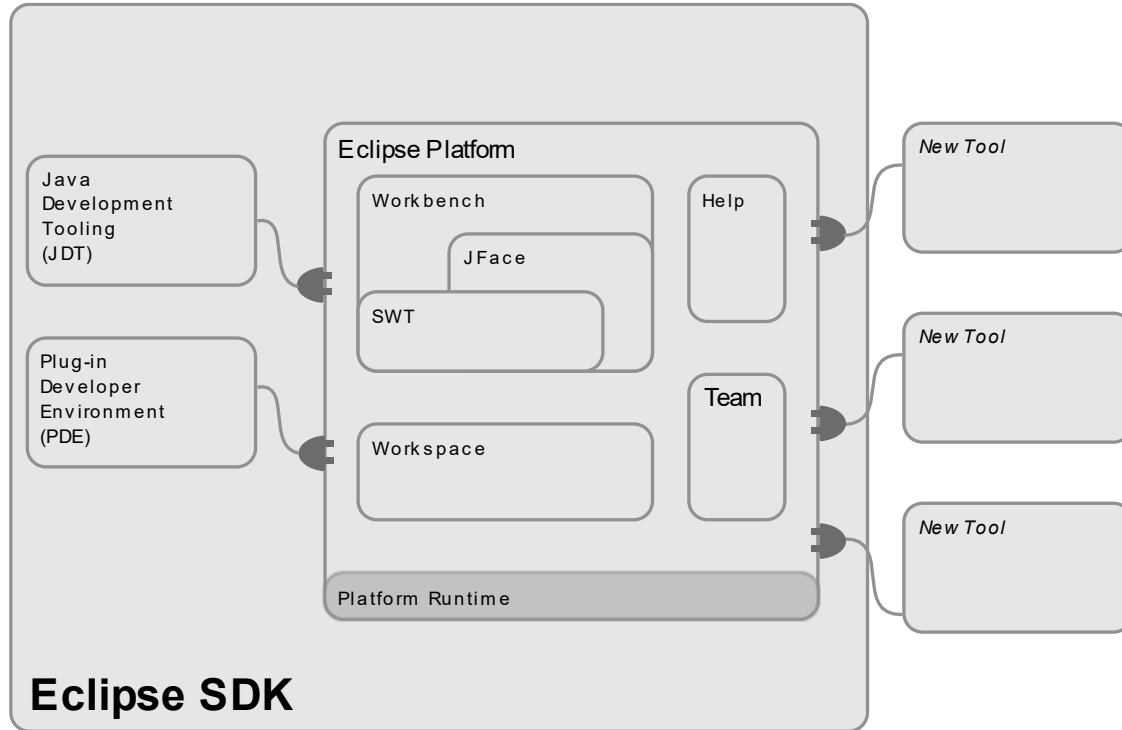
- **Modifiability/maintainability**

- Component can be independent; they do not know to know of the existence of other components
- All data can be managed consistently, as it is managed in one place

Microkernel (Plugin-In) Architecture



Microkernel Architecture: Eclipse



Microkernel Architecture: Web Browsers

chrome web store

Discover Extensions Themes

Search extensions and themes

Filter by All

Sort by Most relevant

Developer Tools

Productivity

Communication

Developer Tools

Education

Tools

Workflow & Planning

Lifestyle

Art & Design

Entertainment

Games

Household

Just for Fun

News & Weather

Shopping

Social Networking

Travel

Well-being

Make Chrome Yours

Accessibility

Functionality & UI

Privacy & Security

GoFullPage - Full Page...
4.9 ★ (77.8K) ⓘ
Capture a screenshot of your current page in entirety and...

ColorZilla
4.6 ★ (3.9K) ⓘ
Advanced Eyedropper, Color Picker, Gradient Generator and...

JSON Formatter
4.6 ★ (1.9K) ⓘ
Makes JSON easy to read. Open source.

User-Agent Switcher for...
3.9 ★ (2.6K) ⓘ
Spoofs & Mimics User-Agent strings.

Lighthouse
4.4 ★ (319) ⓘ
Lighthouse is an open-source, automated tool for improving the...

Similarweb - Website Traff...
4.6 ★ (3.3K) ⓘ
Instant website analysis and SEO metrics at your fingertips.

SEOquake
4.5 ★ (2.5K) ⓘ
SEOquake is a free plugin that provides you with key SEO...

Clear Cache
4.5 ★ (1.1K) ⓘ
Powerful, user-friendly browser data management, right from yo...

SEO META in 1 CLICK
4.9 ★ (1.1K) ⓘ
Displays all meta data and main SEO information for the best SEO

Selenium IDE
3.5 ★ (126) ⓘ
Selenium Record and Playback tool for ease of getting...

ModHeader - Modify HTTP...
3.2 ★ (1.1K) ⓘ
Modify HTTP request headers, response headers, and redirect...

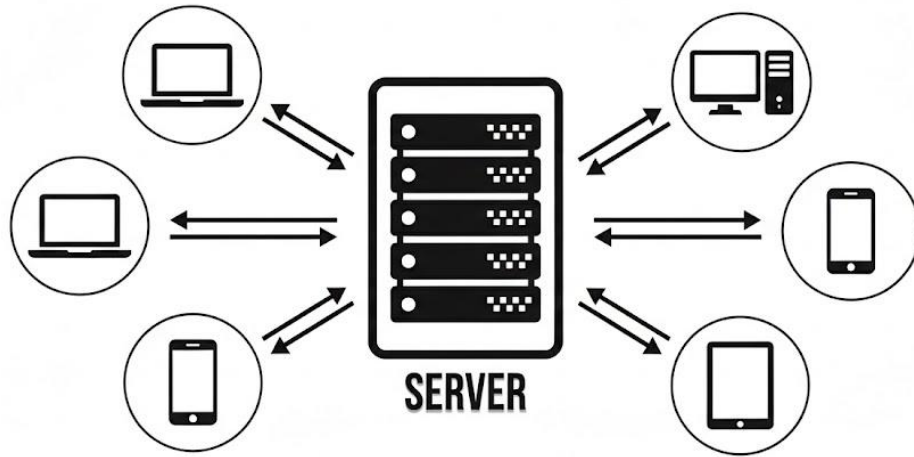
Postman Interceptor
4.3 ★ (957) ⓘ
Capture requests from any website and send them to...



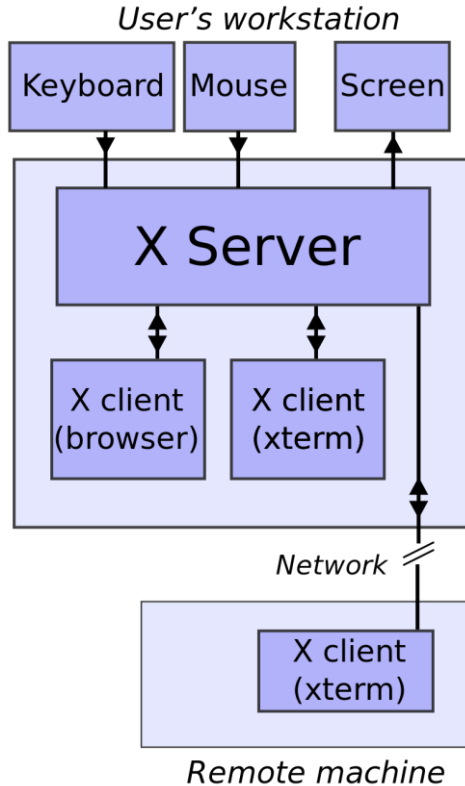
Microkernel Architecture

- Supports modifiability, extensibility, and testability
 - Plug-ins provide a controlled mechanism to extend a core product
 - The plug-ins can be developed (and tested) by different teams
 - The plug-ins can evolve independently from the microkernel, as long as the interfaces do not change

Client-server Architecture



Client-server Architecture





Client-server Architecture: Benefits

- **Scalability:** clients can be independently scaled



Client-server Architecture: Benefits

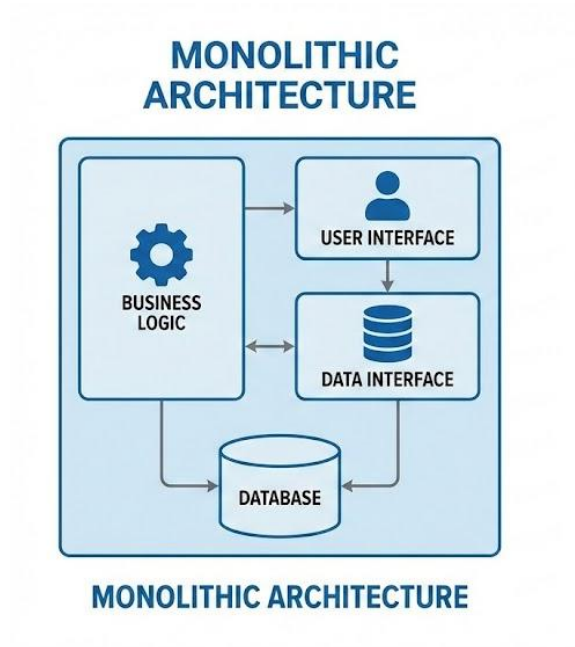
- **Scalability:** clients can be independently scaled
- **Interoperability:** clients can be written in different languages



Client-server Architecture: Benefits

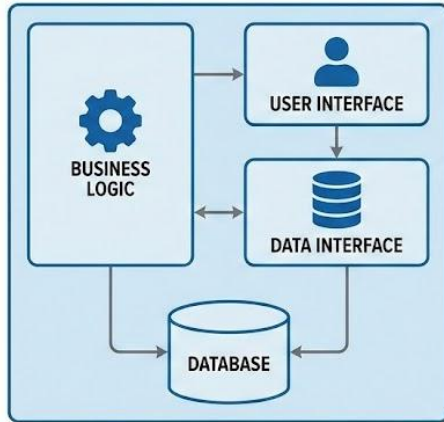
- **Scalability:** clients can be independently scaled
- **Interoperability:** clients can be written in different languages
- **Modifiability/maintainability:** clients and servers can evolve independently

Monoliths



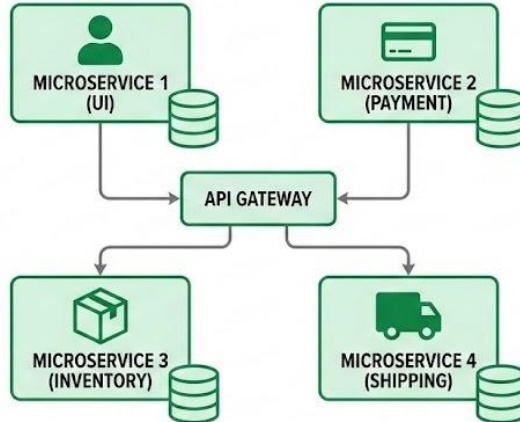
Monoliths

MONOLITHIC ARCHITECTURE



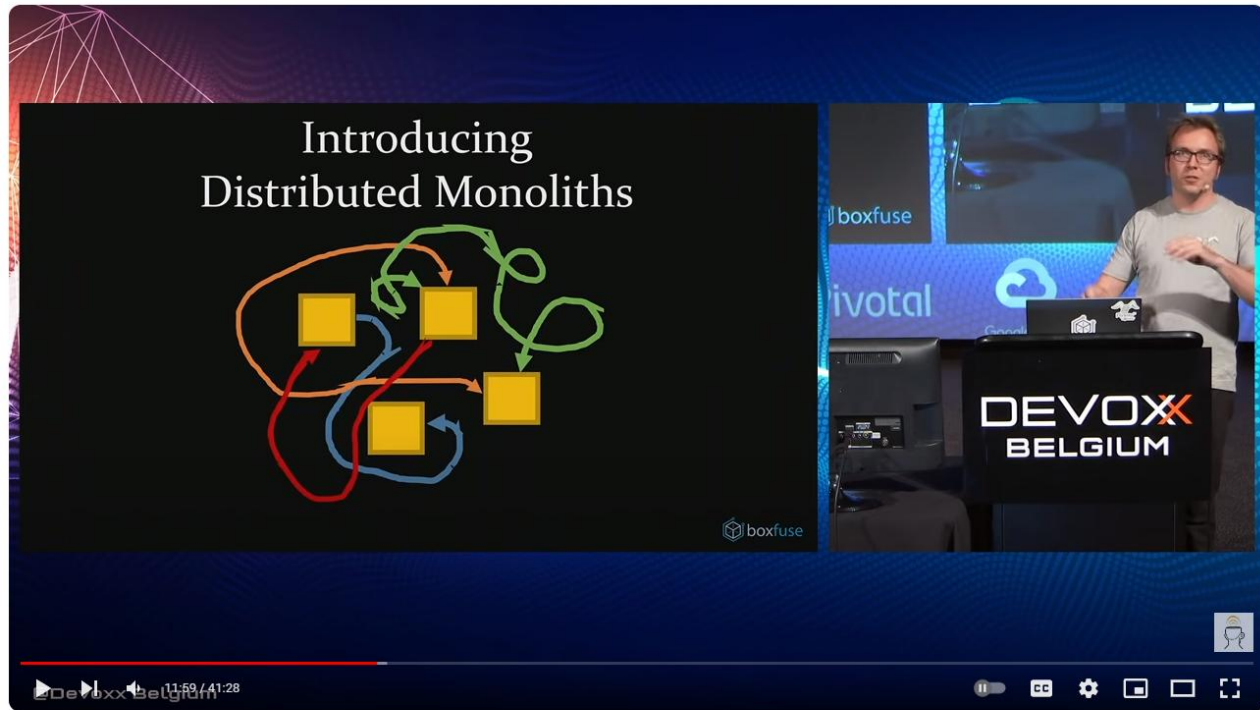
MONOLITHIC ARCHITECTURE

MICROSERVICES ARCHITECTURE



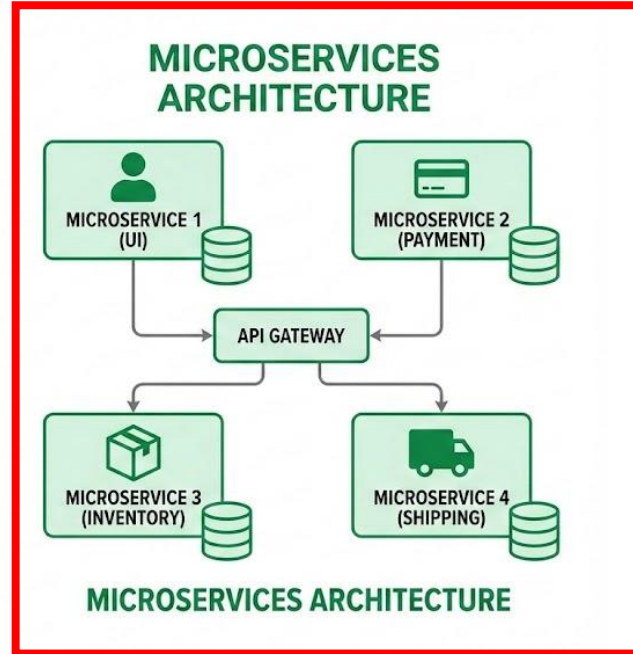
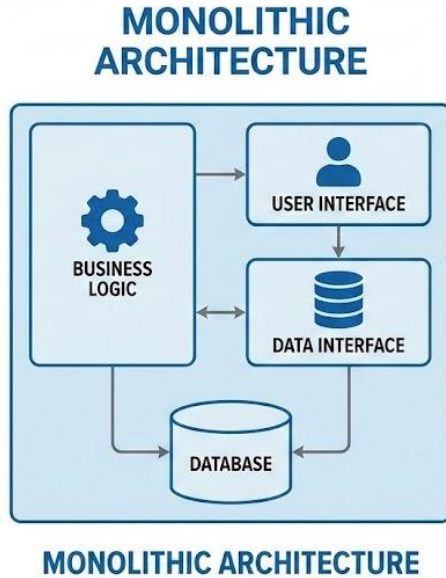
MICROSERVICES ARCHITECTURE

Monoliths: Modular Monolith



Majestic Modular Monoliths by Axel Fontaine

Microservice Architecture



Amazon's Two Pizza Rule



Microservice Architecture

Hybrid Microservice

The diagram illustrates a hybrid microservice architecture. At the top, a 'Client Application' box contains a 'Client Library' box, which in turn contains 'EVCache Client' and 'Service Client' components. Below the client library, there are two main paths: one leading to a set of 'EVCache' (EV) nodes and another leading to a set of 'Service' (S) nodes. The 'EVCache' nodes are represented by red cylinders with 'EV' on them. The 'Service' nodes are represented by white squares with 'S' on them. Below the 'Service' nodes, there is a set of 'Database' (DB) nodes, represented by white cylinders with 'DB' on them. Arrows indicate the flow of data and requests from the client application through the cache and service layers to the databases.

Filmed at
QCon San Francisco 2016

Brought to you by
InfoQ

Mastering Chaos - A Netflix Guide to Microservices

InfoQ
227K subscribers

Subscribe

35K 30:39 / 53:13 • Hybrid Microservice >

Like Comment Share Download Clip Save



Microservice Architecture

- **Deployability** and a quick time to market



Microservice Architecture

- **Deployability** and a quick time to market
- **Independence**: each team can make its own technology choices



Microservice Architecture

- **Deployability** and a quick time to market
- **Independence**: each team can make its own technology choices
- **Scalability**: service instances can be dynamically added

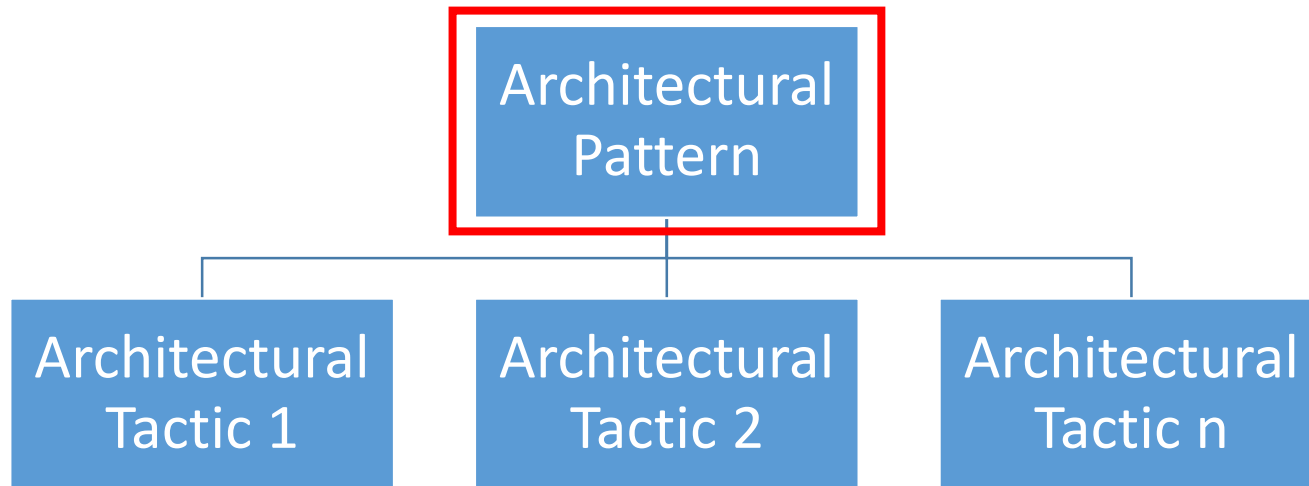


Other Architectural Patterns

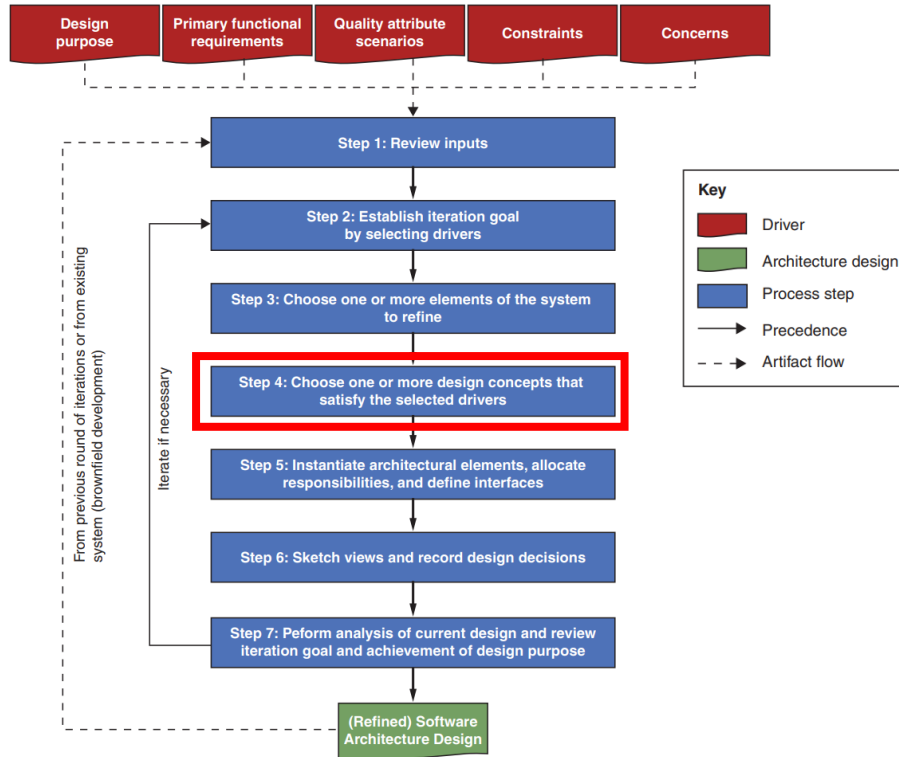
- Service-oriented Architecture (SOA)
- Event-driven Architecture
- Peer-to-peer
- ...

Architectural Tactics and Patterns

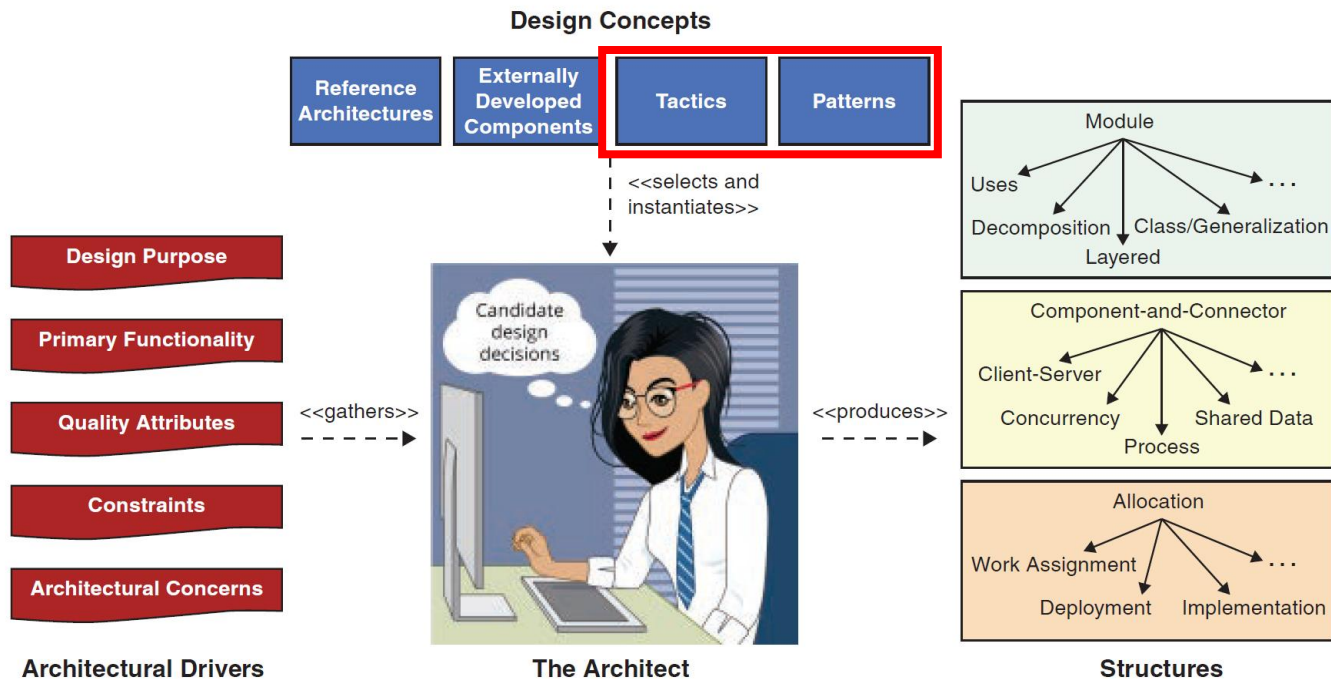
typically comprise multiple architectural tactics



Attribute-Driven Design (ADD)



Approaching Architecture





Summary and Key Points

- Tactics and patterns as design constructs in ADD
- Architectural patterns
 - Layered architecture
 - Pipe-and-filter architecture
 - Model-centered architecture
 - Microkernel architecture
 - Client-server architecture
 - (Modular) Monolith
 - Microservice architecture

Architecture: Sources and References

