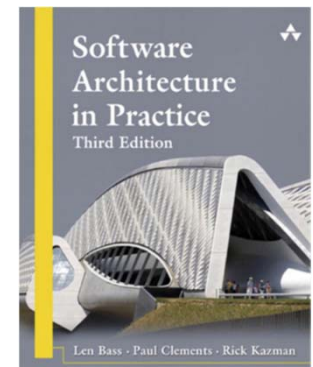


Tactics

Design Concepts 4. Tactics

- **Tactics** are the building blocks of design and the raw materials, from which patterns, frameworks, and styles are constructed.
 - **Techniques** that architects have been using for years to manage quality attribute response goals
 - Design decisions that influence the control of a quality attribute response.
 - Building blocks of architectural patterns
- If architects decides to use a tactics for a quality attribute, then a corresponding architecture should be accompanied.
 - **Availability**
 - **Interoperability**
 - **Modifiability**
 - **Performance**
 - **Security**
 - **Testability**
 - **Usability**



Tactics for Availability

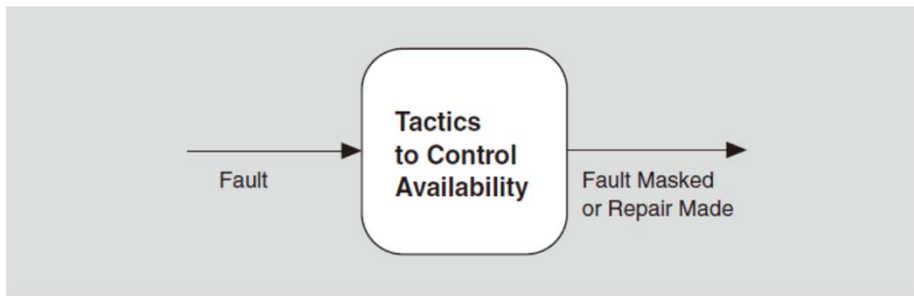
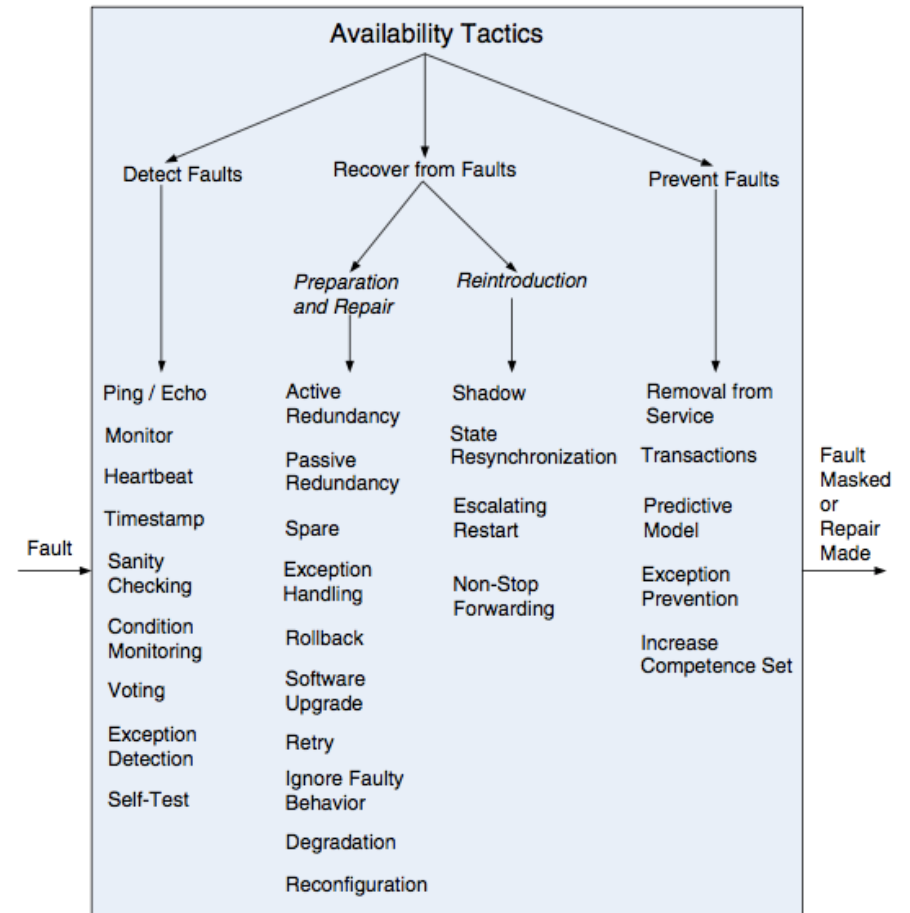


FIGURE 5.4 Goal of availability tactics



Tactics for Availability

- **Detect Faults**

- *Ping/echo*: An asynchronous request/response message pair exchanged between nodes is used to determine reachability and the round-trip delay through the associated network path.
- *Monitor*: A component is used to monitor the state of health of other parts of the system. A system monitor can detect failure or congestion in the network or other shared resources, such as from a denial-of-service attack.
- *Heartbeat*: A periodic message exchange occurs between a system monitor and a process being monitored.
- *Timestamp*: Detect incorrect sequences of events, primarily in distributed message-passing systems.
- *Sanity checking*: Check the validity or reasonableness of a component's operations or outputs; typically based on a knowledge of the internal design, the state of the system, or the nature of the information under scrutiny.
- *Condition monitoring*: Check conditions in a process or device, or validates assumptions made during the design.
- *Voting*: Check that replicated components are producing the same results. Comes in various flavors, such as replication, functional redundancy, analytic redundancy.
- *Exception detection*: Detect a system condition that alters the normal flow of execution, such as a system exception, parameter fence, parameter typing, or timeout.
- *Self-test*: Procedure for a component to test itself for correct operation.

Tactics for Availability

- **Recover from Faults (Preparation and Repair)**

- *Active redundancy (hot spare)*: All nodes in a protection group receive and process identical inputs in parallel, allowing redundant spare(s) to maintain synchronous state with the active node(s).
- *Passive redundancy (warm spare)*: Only the active members of the protection group process input traffic; one of their duties is to provide the redundant spare(s) with periodic state updates.
- *Spare (cold spare)*: Redundant spares of a protection group remain out of service until a failover occurs, at which point a power-on-reset procedure is initiated on the redundant spare prior to its being placed in service.
- *Exception handling*: Deal with the exception by reporting it or handling it, potentially masking the fault by correcting the cause of the exception and retrying.
- *Rollback*: Revert to a previous known good state, referred to as the “rollback line.”
- *Software upgrade*: Perform in-service upgrades to executable code images in a non-service-affecting manner.
- *Retry*: When a failure is transient, retrying the operation may lead to success.
- *Ignore faulty behavior*: Ignore messages sent from a source when it is determined that those messages are spurious.
- *Degradation*: Maintain the most critical system functions in the presence of component failures, dropping less critical functions.
- *Reconfiguration*: Reassign responsibilities to the resources that continue to function, while maintaining as much functionality as possible.

Tactics for Availability

- **Recover from Faults (Reintroduction)**

- *Shadow*: Operate a previously failed or in-service upgraded component in a “shadow mode” for a predefined time prior to reverting the component back to an active role.
- *State resynchronization*: Passive redundancy; state information is sent from active to standby components, in this partner tactic to active redundancy.
- *Escalating restart*: Recover from faults by varying the granularity of the component(s) restarted and minimizing the level of service affected.
- *Non-stop forwarding*: Functionality is split into supervisory and data variants. If a supervisor fails, a router continues forwarding packets along known routes while protocol information is recovered and validated.

- **Prevent Faults**

- *Removal from service*: Temporarily place a system component in an out-of-service state for the purpose of mitigating potential system failures.
- *Transactions*: Bundle state updates so that asynchronous messages exchanged between distributed components are atomic, consistent, isolated, and durable.
- *Predictive model*: Monitor the state of health of a process to ensure that the system is operating within nominal parameters; take corrective action when conditions are detected that are predictive of likely future faults.
- *Exception prevention*: Prevent system exceptions from occurring by masking a fault, or prevent them via smart pointers, abstract data types, and wrappers.
- *Increase competence set*: Design a component to handle more cases, *i.e.*, faults as part of its normal operation.

Tactics for Interoperability

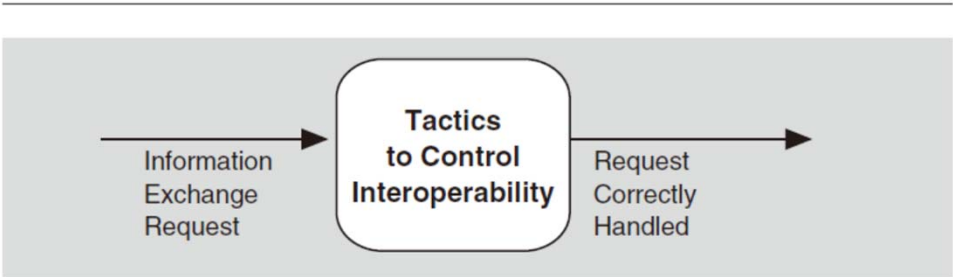
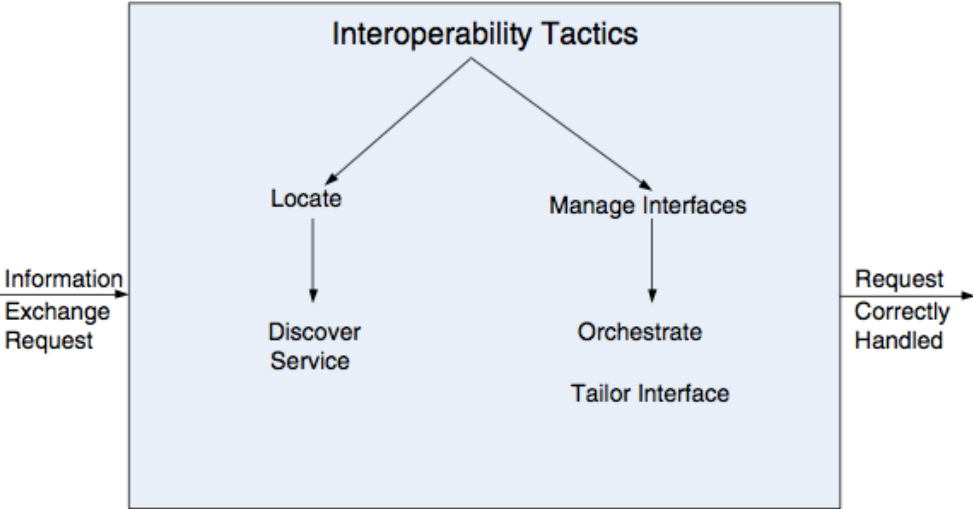


FIGURE 6.2 Goal of interoperability tactics



Tactics for Interoperability

- **Locate**

- *Discover service*: Locate a service by searching a known directory service. There may be multiple levels of indirection in this location process - that is, a known location may point to another location that in turn can be searched for the service.

- **Manage Interfaces**

- *Orchestrate*: Use a control mechanism to coordinate, manage, and sequence the invocation of services. Orchestration is used when systems must interact in a complex fashion to accomplish a complex task.
- *Tailor interface*: Add or remove capabilities to an interface such as translation, buffering, or data smoothing.

Tactics for Modifiability

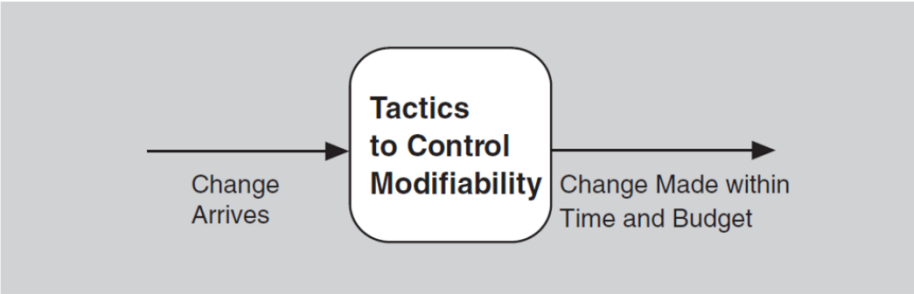
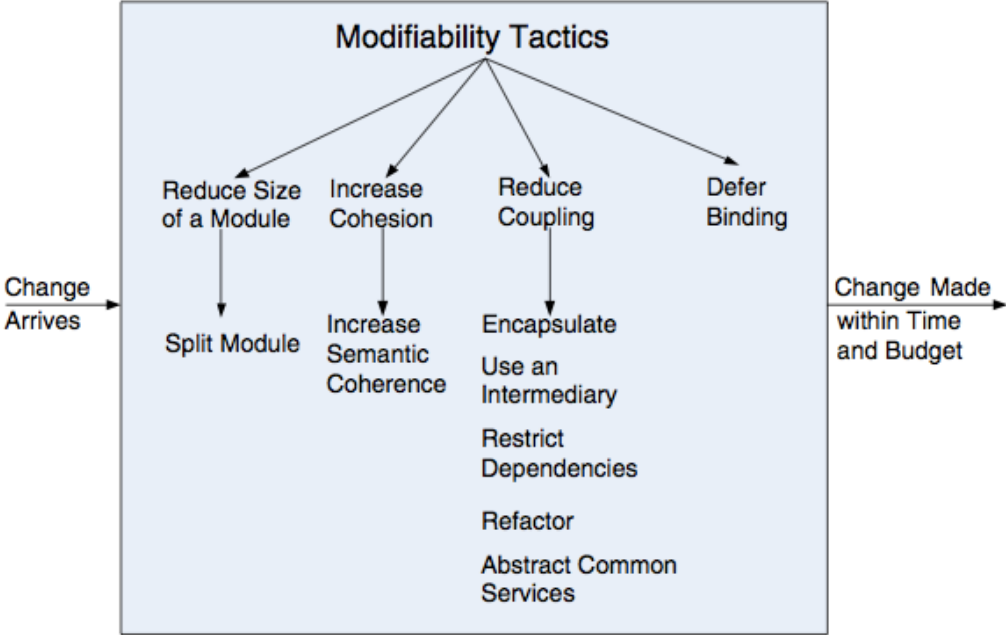


FIGURE 7.2 The goal of modifiability tactics



Tactics for Modifiability

- **Reduce Size of a Module**

- *Split module*: If the module being modified includes a great deal of capability, the modification costs will likely be high. Refining the module into several smaller modules will reduce the average cost of future changes.

- **Increase Cohesion**

- *Increase semantic coherence*: If the responsibilities A and B in a module do not serve the same purpose, they should be placed in different modules. This may involve creating a new module or moving a responsibility to an existing module.

- **Reduce Coupling**

- *Encapsulate*: Encapsulation introduces an explicit interface to a module. This interface includes an API and its associated responsibilities, such as “perform a syntactic transformation on an input parameter to an internal representation.”
- *Use an intermediary*: Given a dependency between responsibility A and responsibility B (for example, carrying out A first requires carrying out B), the dependency can be broken by using an intermediary.
- *Restrict dependencies*: Restrict the modules that a given module interacts with or depends on.
- *Refactor*: Refactoring is undertaken when two modules are affected by the same change because they are (at least partial) duplicates of each other.
- *Abstract common services*: When two modules provide not quite the same but similar services, it may be cost-effective to implement the services just once in a more general (abstract) form.

- **Defer Binding**

- *Defer binding*: Allow decisions to be bound after development time.

Tactics for Performance

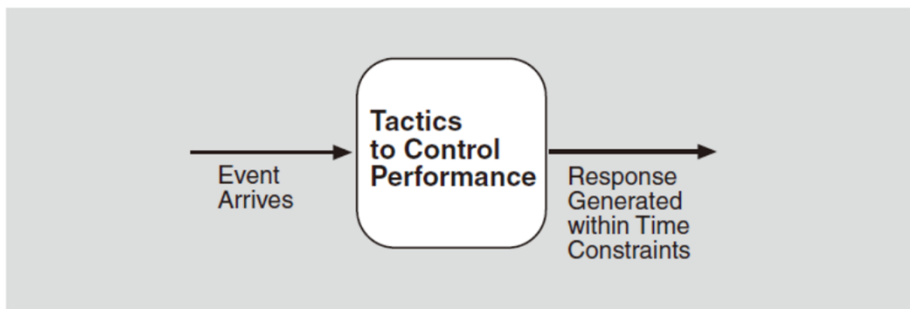
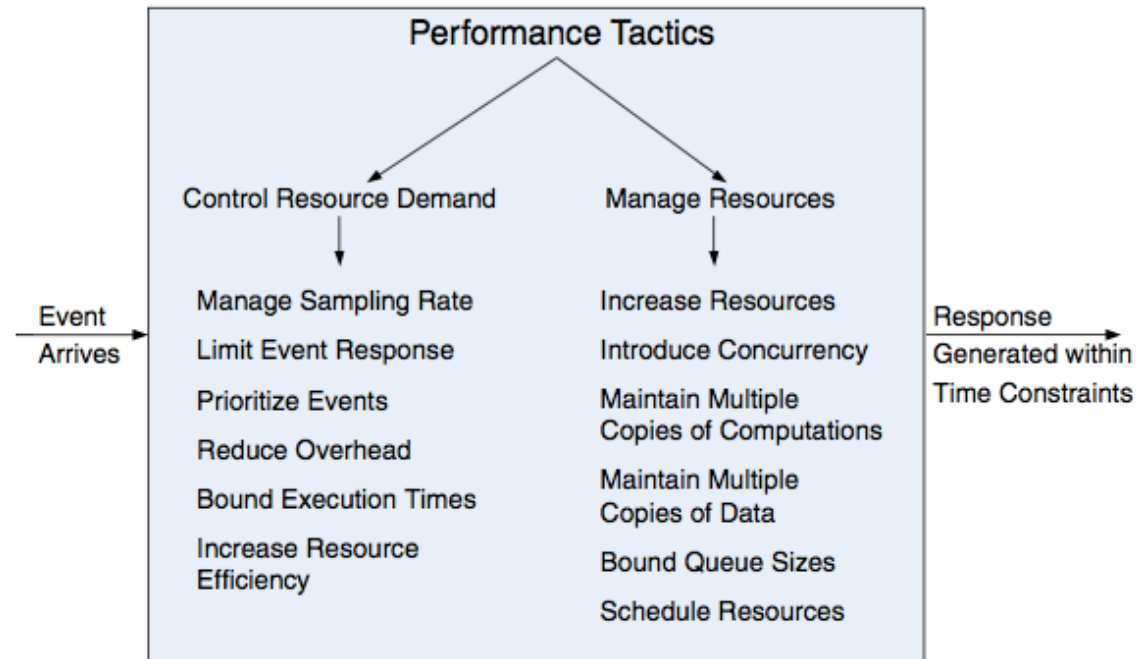


FIGURE 8.2 The goal of performance tactics



Tactics for Performance

- **Control Resource Demand**

- *Manage sampling rate*: If it is possible to reduce the sampling frequency at which a stream of data is captured, then demand can be reduced, albeit typically with some loss of fidelity.
- *Limit event response*: Process events only up to a set maximum rate, thereby ensuring more predictable processing when the events are actually processed.
- *Prioritize events*: If not all events are equally important, you can impose a priority scheme that ranks events according to how important it is to service them.
- *Reduce overhead*: The use of intermediaries (important for modifiability) increases the resources consumed in processing an event stream; removing them improves latency.
- *Bound execution times*: Place a limit on how much execution time is used to respond to an event.
- *Increase resource efficiency*: Improving the algorithms used in critical areas will decrease latency.

- **Manage Resources**

- *Increase resources*: Faster processors, additional processors, additional memory, and faster networks all have the potential to reduce latency.
- *Increase concurrency*: If requests can be processed in parallel, the blocked time can be reduced. Concurrency can be introduced by processing different streams of events on different threads or by creating additional threads to process different sets of activities.
- *Maintain multiple copies of computations*: The purpose of replicas is to reduce the contention that would occur if all computations took place on a single server.
- *Maintain multiple copies of data*: Keep copies of data (with one potentially being a subset of the other) on storage with different access speeds.
- *Bound queue sizes*: Control the maximum number of queued arrivals and consequently the resources used to process the arrivals.
- *Schedule resources*: When there is contention for a resource, the resource must be scheduled.

Tactics for Security

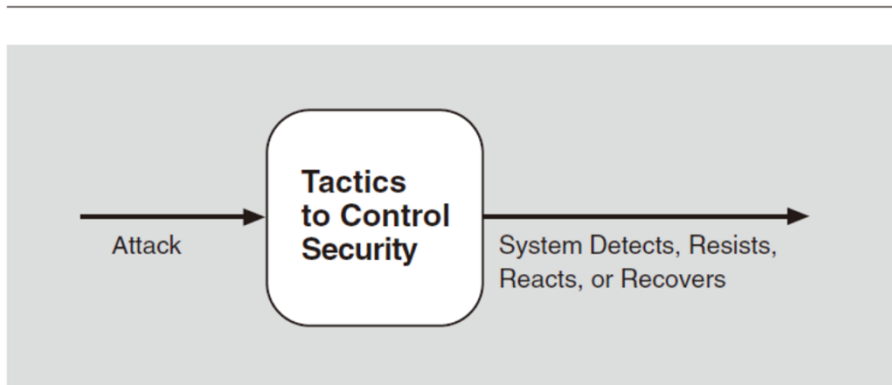
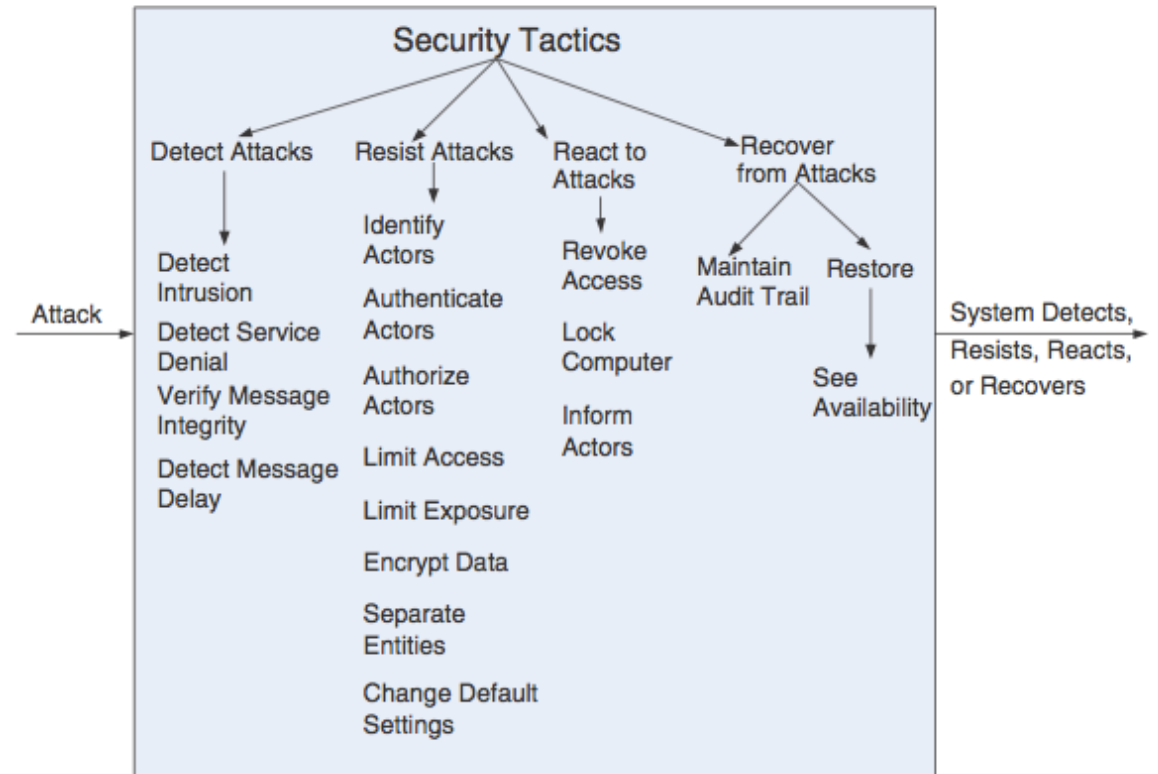


FIGURE 9.2 The goal of security tactics



Tactics for Security

- **Detect Attacks**

- *Detect intrusion*: Compare network traffic or service request patterns within a system to a set of signatures or known patterns of malicious behavior stored in a database.
- *Detect service denial*: Compare the pattern or signature of network traffic coming into a system to historic profiles of known denial-of-service attacks.
- *Verify message integrity*: Use techniques such as checksums or hash values to verify the integrity of messages, resource files, deployment files, and configuration files.
- *Detect message delay*: By checking the time that it takes to deliver a message, it is possible to detect suspicious timing behavior.

- **React to Attacks**

- *Revoke access*: Limit access to sensitive resources, even for normally legitimate users and uses, if an attack is suspected.
- *Lock computer*: Limit access to a resource if there are repeated failed attempts to access it.
- *Inform actors*: Notify operators, other personnel, or cooperating systems when an attack is suspected or detected.

Tactics for Security

- **Resist Attacks**

- *Identify actors*: Identify the source of any external input to the system.
- *Authenticate actors*: Ensure that an actor (user or a remote computer) is actually who or what it purports to be.
- *Authorize actors*: Ensure that an authenticated actor has the rights to access and modify either data or services.
- *Limit access*: Control what and who may access which parts of a system, such as processors, memory, and network connections.
- *Limit exposure*: Reduce the probability of a successful attack, or restrict the amount of potential damage—for example, by concealing facts about a system (“security by obscurity”) or by dividing and distributing critical resources (“don’t put all your eggs in one basket”).
- *Encrypt data*: Apply some form of encryption to data and to communication.
- *Validate input*: Validate input from a user or an external system before accepting it in the system.
- *Separate entities*: Use physical separation on different servers attached to different networks, virtual machines, or an “air gap.”
- *Change default settings*: Force the user to change settings assigned by default.

- **Recover from Attacks**

- In addition to the *Availability Tactics* for recovery of failed resources, an *Audit* may be performed to recover from attacks.
- *Maintain Audit Trail*: Keep a record of user and system actions and their effects, to help trace the actions of, and to identify, an attacker.

Tactics for Testability

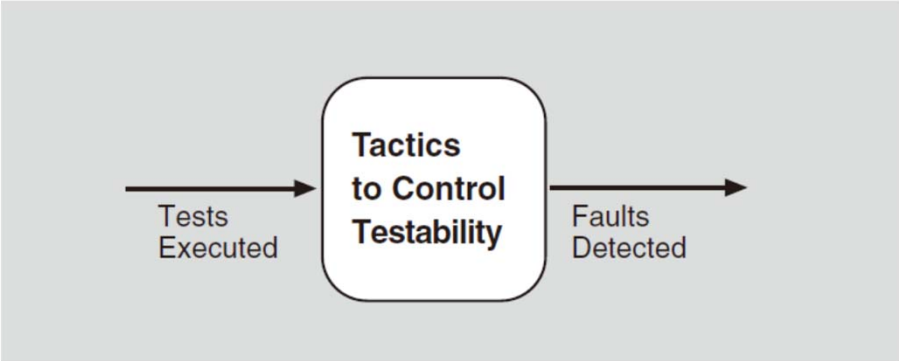
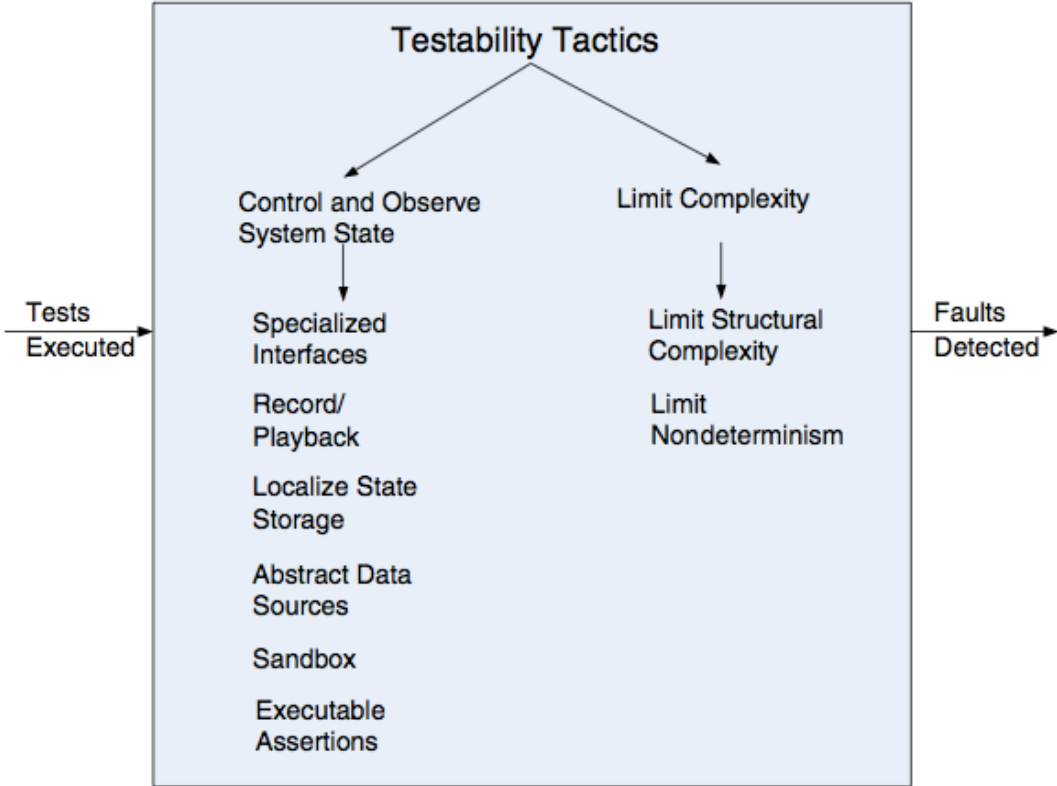


FIGURE 10.3 The goal of testability tactics



Tactics for Testability

- **Control and Observe System State**

- *Specialized interfaces*: Control or capture variable values for a component either through a test harness or through normal execution.
- *Record/playback*: Capture information crossing an interface and use it as input for further testing.
- *Localize state storage*: To start a system, subsystem, or module in an arbitrary state for a test, it is most convenient if that state is stored in a single place.
- *Abstract data sources*: Abstracting the interfaces lets you substitute test data more easily.
- *Sandbox*: Isolate the system from the real world to enable experimentation that is unconstrained by the worry about having to undo the consequences of the experiment.
- *Executable assertions*: Assertions are (usually) hand-coded and placed at desired locations to indicate when and where a program is in a faulty state.

- **Limit Complexity**

- *Limit structural complexity*: Avoid or resolve cyclic dependencies between components, isolate and encapsulate dependencies on the external environment, and reduce dependencies between components in general.
- *Limit nondeterminism*: Find all the sources of non-determinism, such as unconstrained parallelism, and weed them out as far as possible.

Tactics for Usability

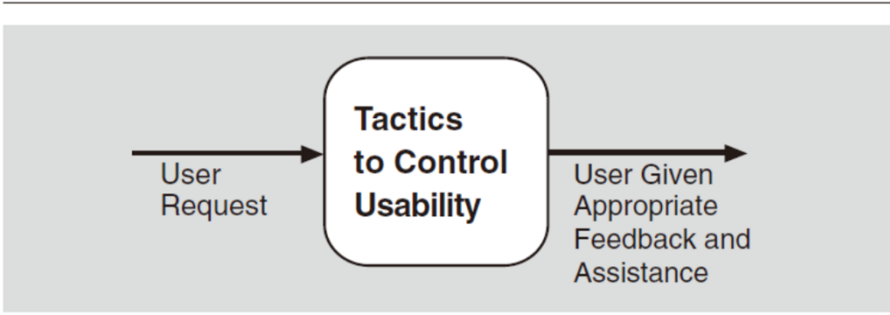
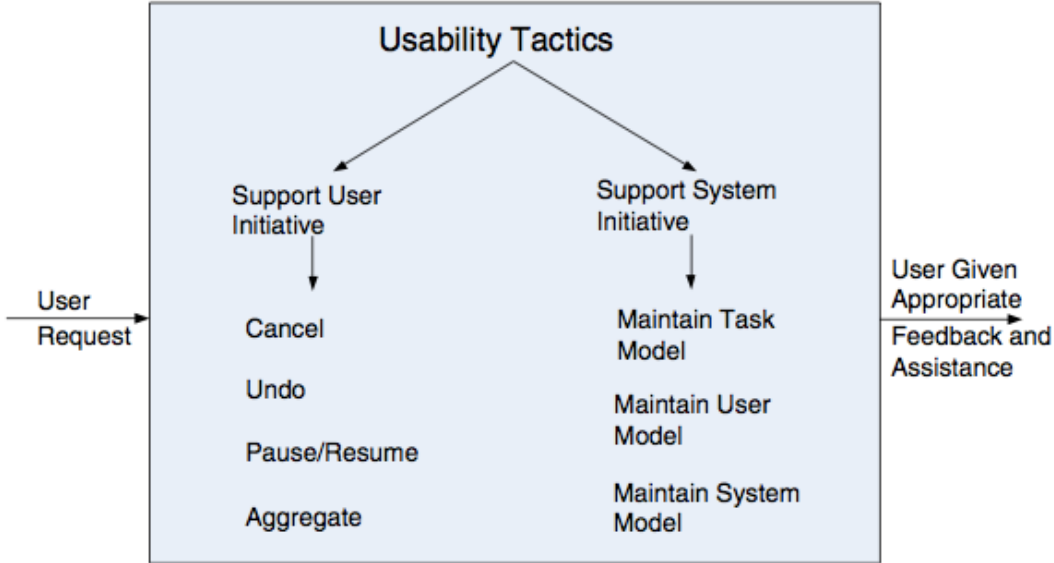


FIGURE 11.2 The goal of runtime usability tactics



Tactics for Usability

- **Support User Initiative**

- *Cancel*: The system must listen for the cancel request; the command being canceled must be terminated; resources used must be freed; and collaborating components must be informed.
- *Pause/resume*: Temporarily free resources so that they may be reallocated to other tasks.
- *Undo*: Maintain a sufficient amount of information about system state so that an earlier state may be restored at the user's request.
- *Aggregate*: Aggregate lower-level objects into a group, so that a user operation may be applied to the group, freeing the user from the drudgery.



- **Support System Initiative**

- *Maintain task model*: Determine the context so the system can have some idea of what the user is attempting and provide assistance.
- *Maintain user model*: Explicitly represent the user's knowledge of the system, the user's behavior in terms of expected response time, and other characteristics of the system.
- *Maintain system model*: The system maintains an explicit model of itself. This tactic is used to determine expected system behavior so that appropriate feedback can be given to the user.

State-of-the-art Researches on Tactics


The Journal of Systems & Software 197 (2023) 111558

Contents lists available at [ScienceDirect](#)

 **The Journal of Systems & Software** 

journal homepage: www.elsevier.com/locate/jss

Architectural tactics in software architecture: A systematic mapping study[☆]



Gastón Márquez^{a,*}, Hernán Astudillo^b, Rick Kazman^c

^a Department of Electronics and Informatics, Universidad Técnica Federico Santa María, Concepción, Chile
^b Department of Informatics, Universidad Técnica Federico Santa María, Santiago, Chile
^c Department of Information Technology Management, University of Hawaii, Honolulu, HI, USA

<p>ARTICLE INFO</p> <p><i>Article history:</i> Received 16 September 2021 Received in revised form 8 November 2022 Accepted 12 November 2022 Available online 22 November 2022</p> <p><i>Keywords:</i> Architectural tactics Systematic mapping study Software architecture Quality attributes</p>	<p>ABSTRACT</p> <p>Architectural tactics are a key abstraction of software architecture, and support the systematic design and analysis of software architectures to satisfy quality attributes. Since originally proposed in 2003, architectural tactics have been extended and adapted to address additional quality attributes and newer kinds of systems, making quite hard for researchers and practitioners to master this growing body of specialized knowledge. This paper presents the design, execution and results of a systematic mapping study of architectural tactics in software architecture literature. The study found 552 studies in well-known digital libraries, of which 79 were selected and 12 more were added with snowballing, giving a total of 91 primary studies. Key findings are: (i) little rigor has been used to characterize and define architectural tactics; (ii) most architectural tactics proposed in the literature do not conform to the original definition; and (iii) there is little industrial evidence about the use of architectural tactics. This study organizes and summarizes the scientific literature to date about architectural tactics, identifies research opportunities, and argues for the need of more systematic definition and description of tactics.</p> <p><i>Editor's note: Open Science material was validated by the Journal of Systems and Software Open Science Board.</i></p> <p>© 2022 Elsevier Inc. All rights reserved.</p>
-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

State-of-the-art Researches on Tactics

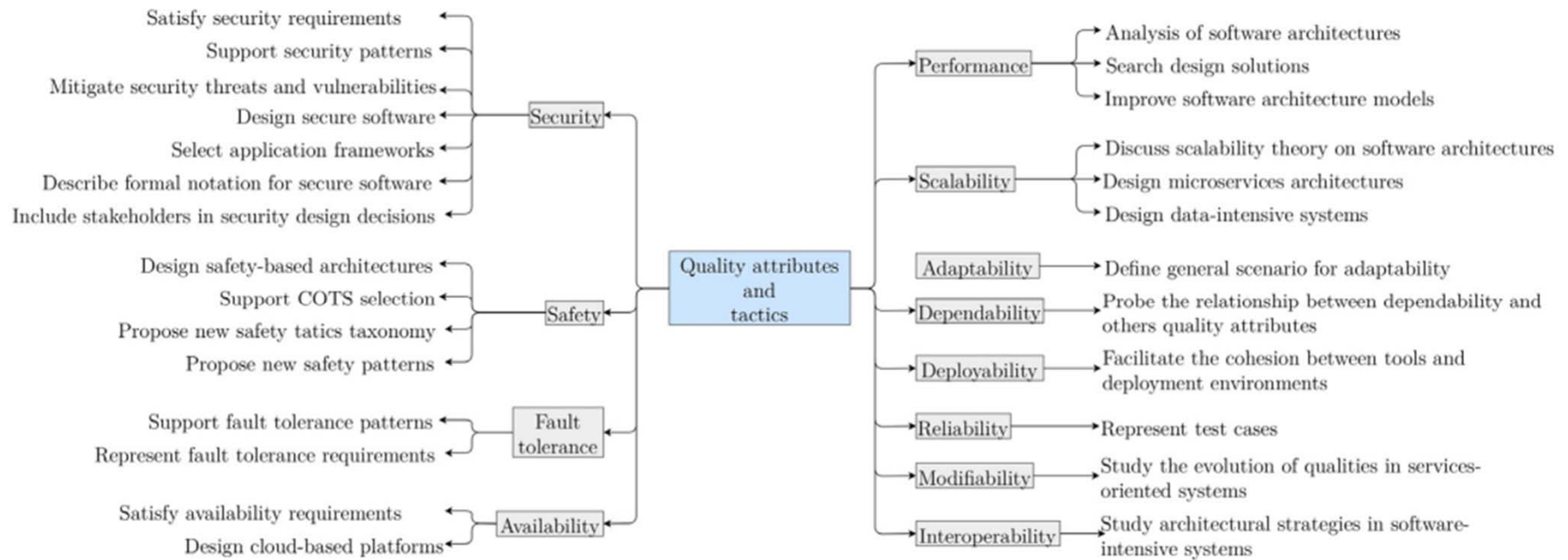


Fig. 10. Taxonomy of purposes to achieve quality attributes using tactics.

State-of-the-art Researches on Tactics

Table 5
Quality attributes addressed by tactics research.

QA	Description	# of studies
Adaptability	Adaptability controls how easy it is to change the system if requirements have changed (Tarvainen, 2008).	1
Dependability	Property of a system that delivers services at a specified reliability level and the system's ability to avoid failures that are serious and numerous (Avizienis et al., 2004).	1
Reliability	The degree to which a system, product or component performs specified functions under specified conditions for a specified period of time (ISO 25000 software and data quality, 2020).	1
Modifiability	The degree to which a product or system can be effectively and efficiently modified without introducing defects or degrading existing product quality (ISO 25000 software and data quality, 2020).	1
Interoperability	The ability of systems to share data and enable the exchange of information and knowledge between them (Bass et al., 2013).	1
Deployability	The time to get code into production after a commit (Bass, 2016).	2
Scalability	This quality attribute represents a system's ability to handle an increasing amount of work, or its potential to be expanded to accommodate growth (Kazman and Kruchten, 2012a).	3
Performance	Performance concerns itself with a software system's ability to meet timing requirements (Bass et al., 2013).	4
Safety	Attention to safety is required at each step of the software development process, identifying which functions are critical to the system's safe functioning and tracing those functions down into the modules that support them (System Safety Engineering, 0000).	4
Availability	Characteristic of architectures that measures the degree to which system resources are available for use by end-users over a given time period (ISO 25000 software and data quality, 2020).	4
Fault tolerance	This quality attribute is related to a system's ability to continue to function continuously in the event of faults (ISO 25000 software and data quality, 2020).	5
Security	The degree to which a product or system protects information and data so that individuals or other systems have the appropriate degree of access to data according to their types and levels of authorization (ISO 25000 software and data quality, 2020).	18

State-of-the-art Researches on Tactics

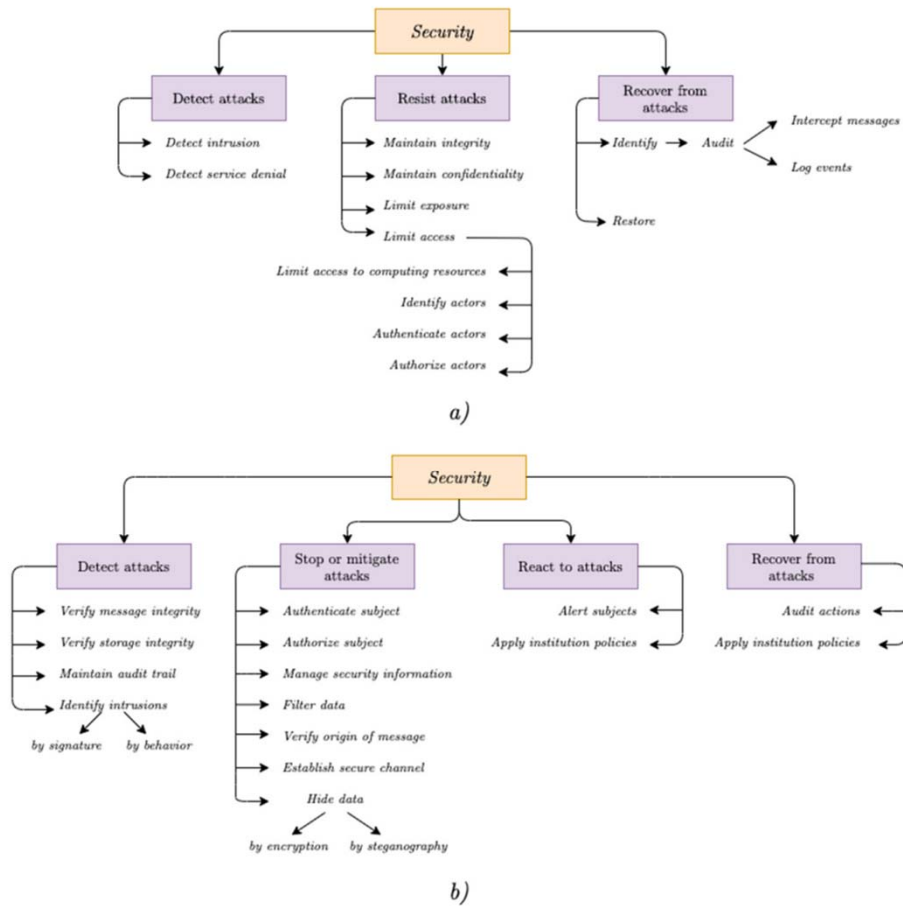


Fig. 11. Security tactics taxonomy proposed by S20 (a) and S48 (b).

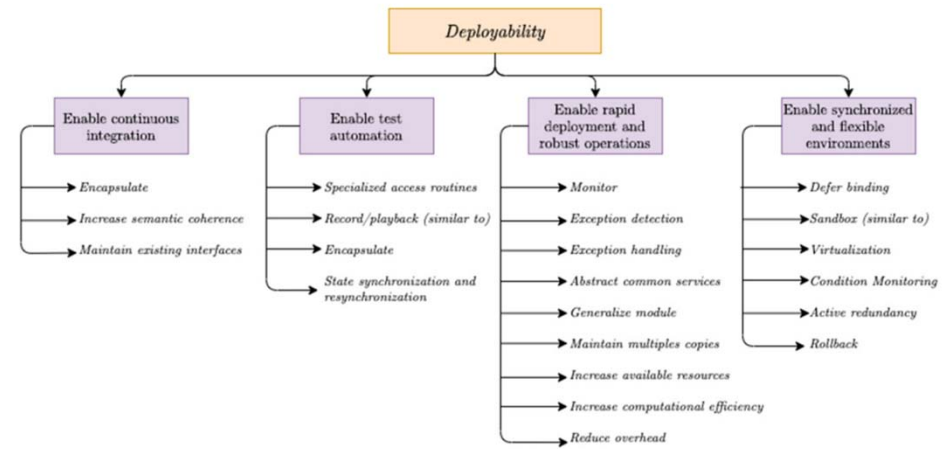


Fig. 12. Deployability tactics taxonomy proposed by S39.

State-of-the-art Researches on Tactics

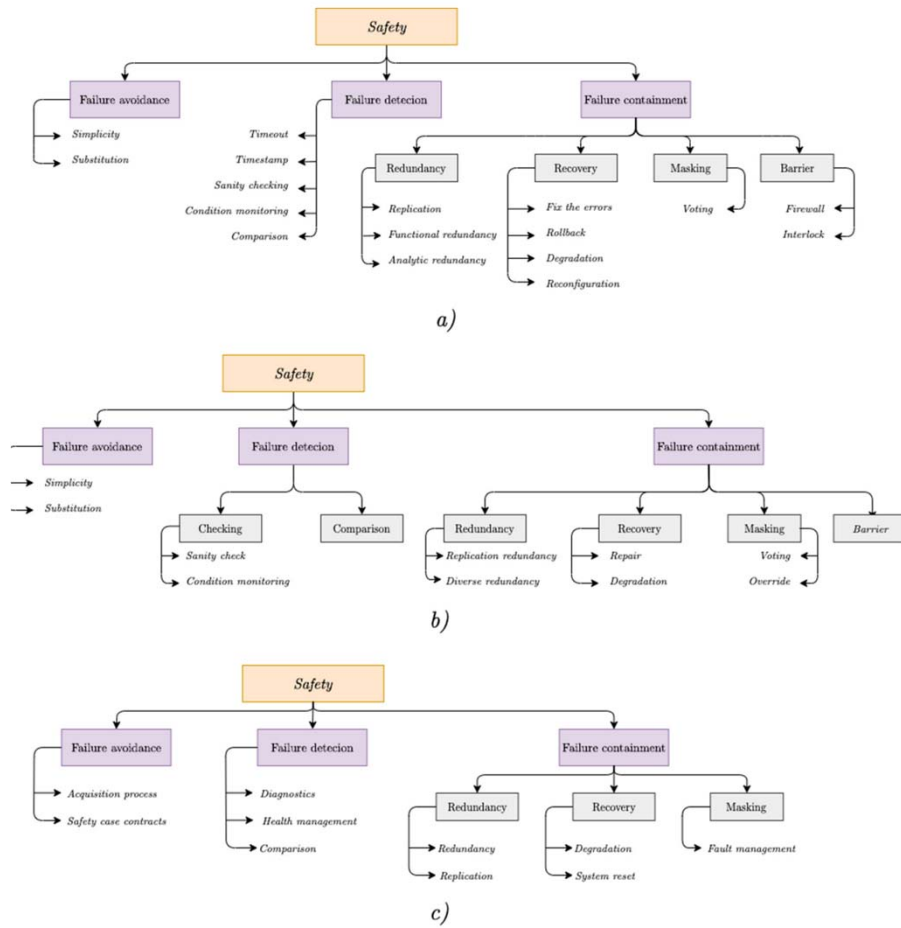


Fig. 13. Safety tactics taxonomy proposed by S2 (a), S30 (b) and S7 (c).

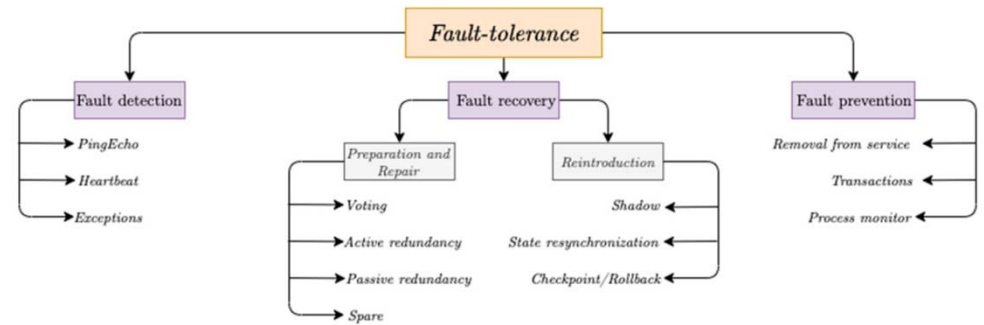


Fig. 14. Fault-tolerance tactics taxonomy proposed by S4.

State-of-the-art Researches on Tactics

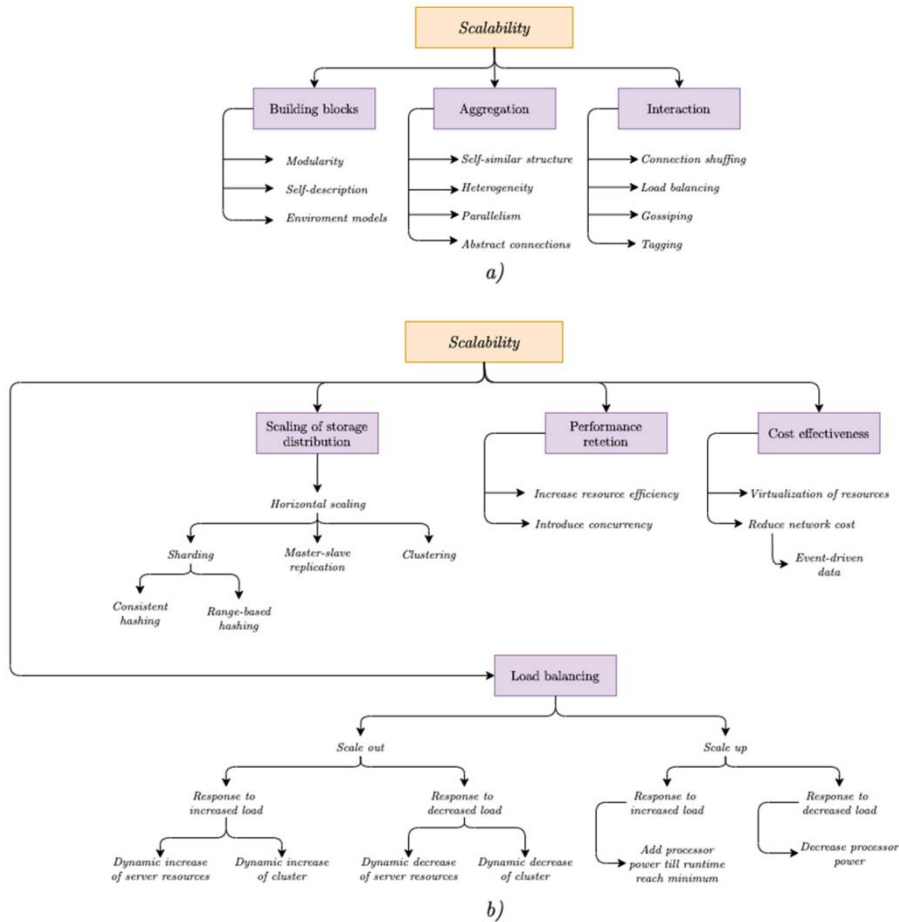


Fig. 15. Scalability tactics taxonomy proposed by S19 (a) and S79 (b).

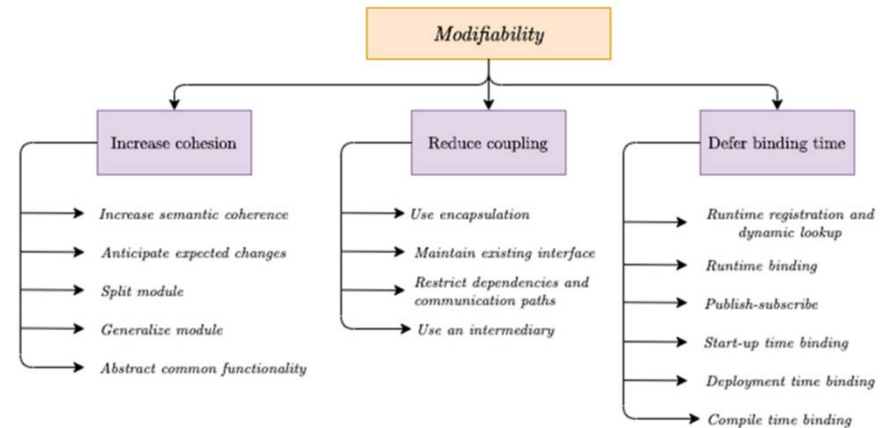


Fig. 16. Modifiability tactics taxonomy proposed by S69.



3. ASR Analysis
> 3.2 QAS > Tactics



4. Architecture Design & Evaluation
> Design Concepts > 4. Tactics