Self-Adaptive Systems

Reference: An Introduction to Self-Adaptive Systems: A Contemporary Software Engineering Perspective, First Edition. Danny Weyns

Key Historical Context

From the earliest days of computing, theorists recognized the potential for computation to change itself (self-modification)

“Traditional” software engineering focuses on getting things right before a system is deployed, known environment, components under the control of the developers

Today systems work in much more uncertain contexts

The software engineering discipline of Self-Adaptive Systems (SAS) attempts to solve this, providing principles and tools to harness the vast potential of adaptation

SAS: A Convergence of Disciplines

SAS draws heavily on knowledge from other fields:

- **Control Theory**: Techniques for maintaining a system's envelope of behavior within desired ranges
- **Biology & Ecology**: Ability of organisms/populations to respond to environmental changes
- **Immunology**: Organic mechanisms of self-healing
- **Software Architecture**: Patterns for structuring systems to enable predictable construction
- **Artificial Intelligence**: Mechanisms supporting autonomy

Defining a SAS

There are two generally acknowledged principles that determine what constitutes a self-adaptive system

External Principle (opaque system)

A self-adaptive system is a **system** that can **handle changes** and uncertainties in its environment, the system itself, and its goals **autonomously** (i.e., without or with minimal required human intervention)

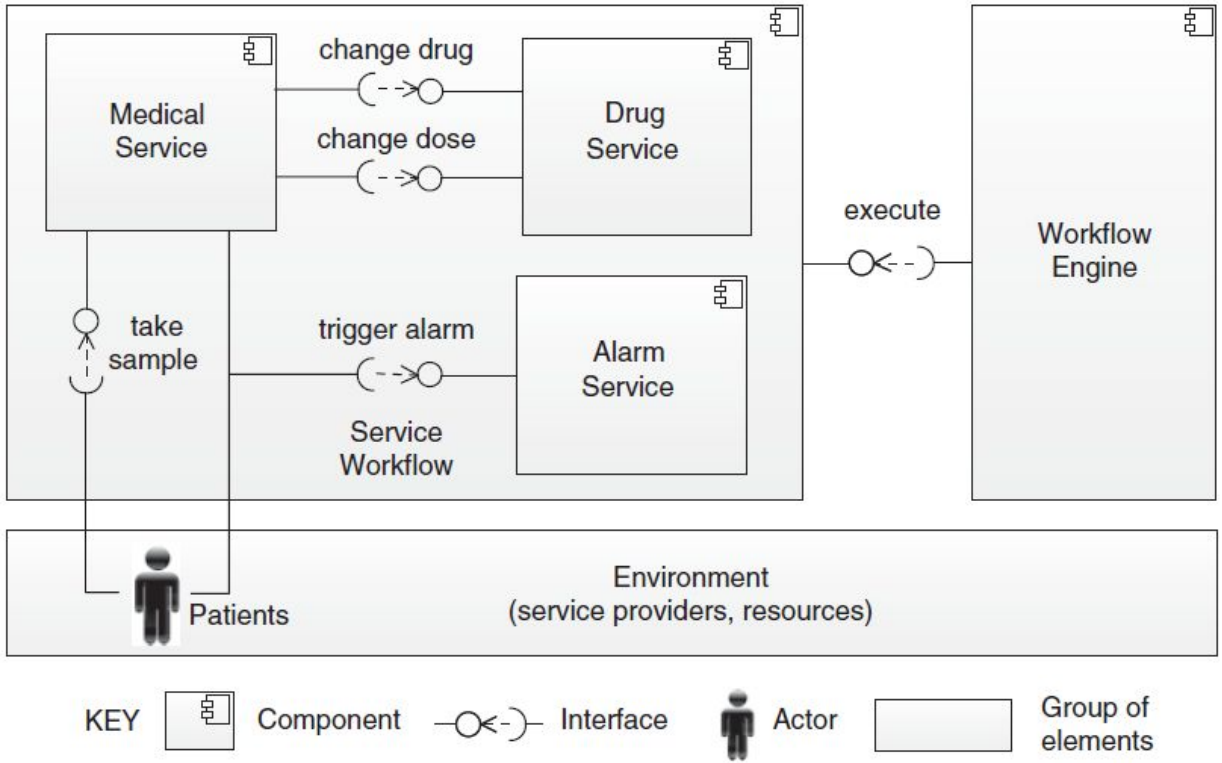
Internal Principle (engineer's view)

A self-adaptive system comprises two distinct parts:

1. **Domain Concerns**: Interacts with the environment, responsible for user goals

2. **Adaptation Concerns**: Consists of a feedback loop that monitors the first part (and its environment) and is responsible for managing the system under changing conditions

SAS: an example

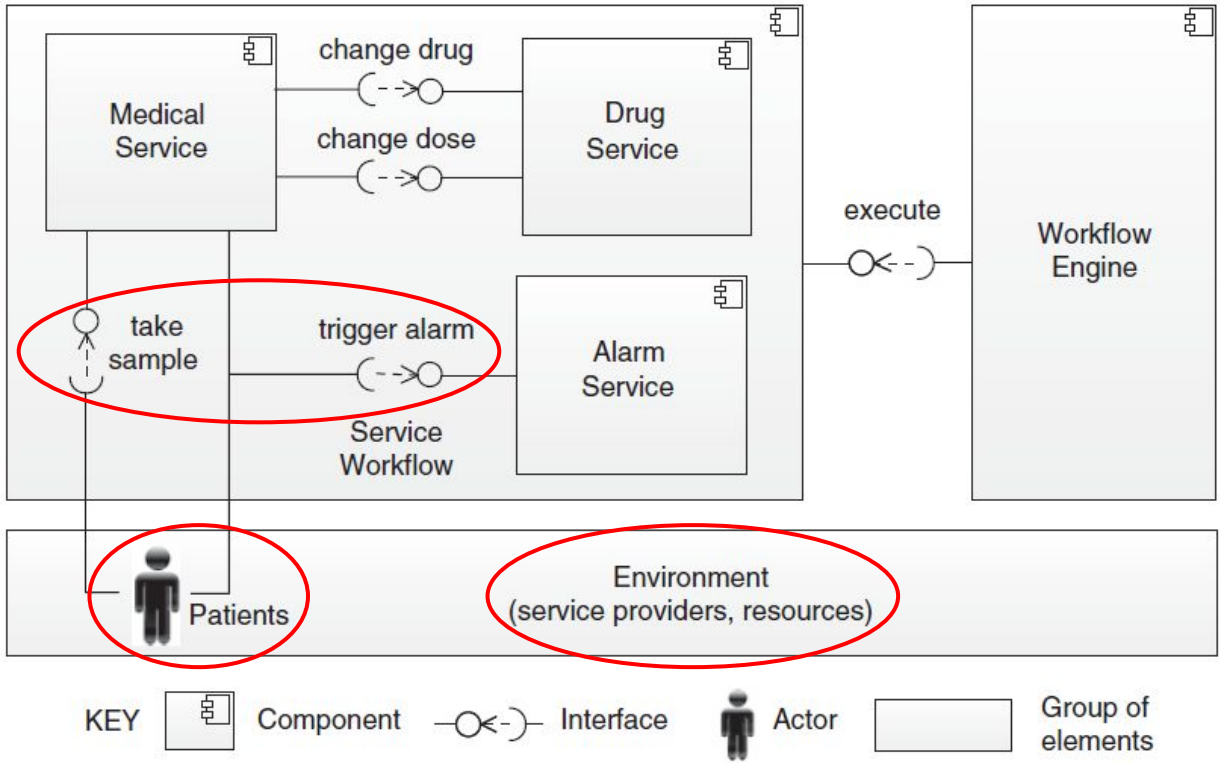


Sources of Uncertainty

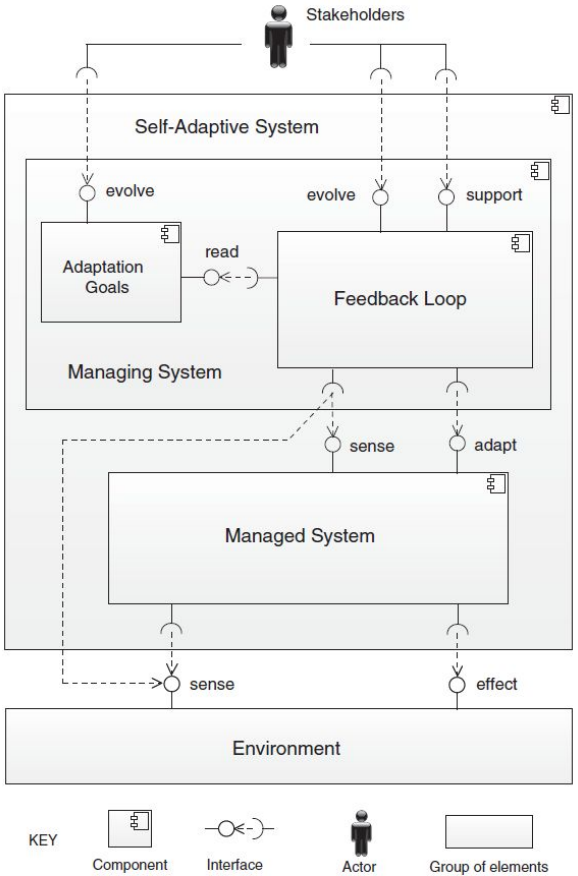
Patient-related variability: differences in health conditions, behaviors, and response times

Workflow dynamics: unpredictable frequency and sequence of service invocations depending on patient states

Service availability: changes in the number or status of third-party service instances over time



SAS Conceptual Model



SAS Conceptual Model: Environment

The environment refers to the **part of the external world with which a self-adaptive system interacts** and in which the **effects of the system will be observed** and evaluated

The environment can be **sensed** and **effected** through sensors and actuators

Uncertainty in terms of **what is sensed** by the sensors and **actuators outcomes**

Environment Example

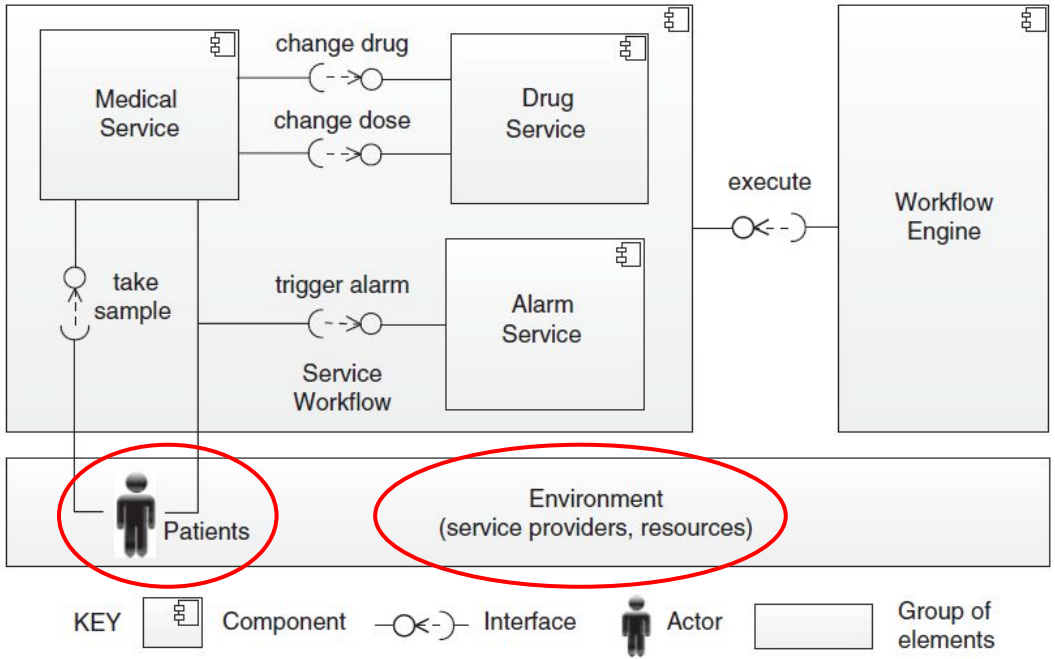
Environment includes:

1) the patients that make use of the system

2) the application devices with the sensors that measure vital parameters of patients

3) the service providers with the services instances they offer

4) the network connections used in the system



SAS Conceptual Model: Managed System

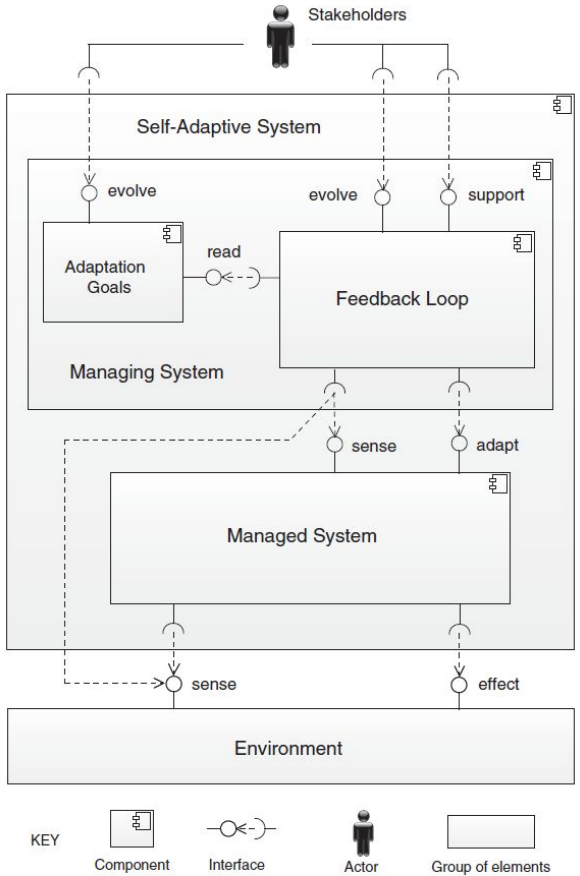
Managed System: application software that **realizes the functions** of the system and **senses/effects** the environment

To support adaptations, the managed system needs to be equipped with **sensors** and **actuators**

Enable **monitoring**

Execute adaptation actions

SAS Conceptual Model



Managed System Example (1)

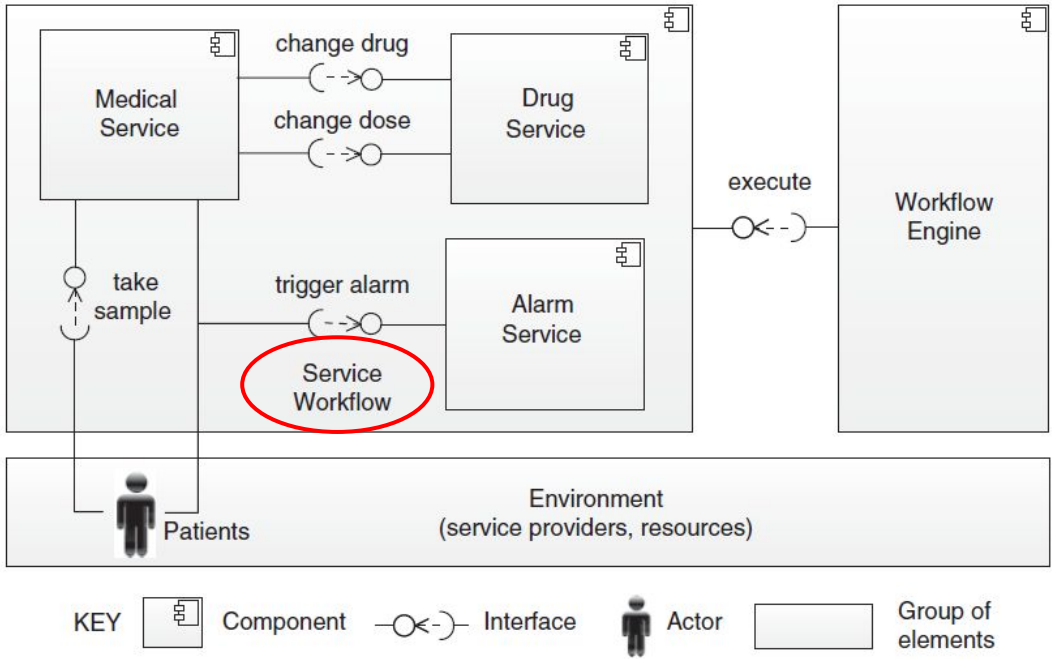
Managed System includes:

1) receives messages from patients with values of their vital parameters

2) invokes a drug service to notify a local pharmacy to deliver new medication

3) invokes an alarm service in case of an emergency to notify medical staff to visit the patient

4) the network connections used in the system

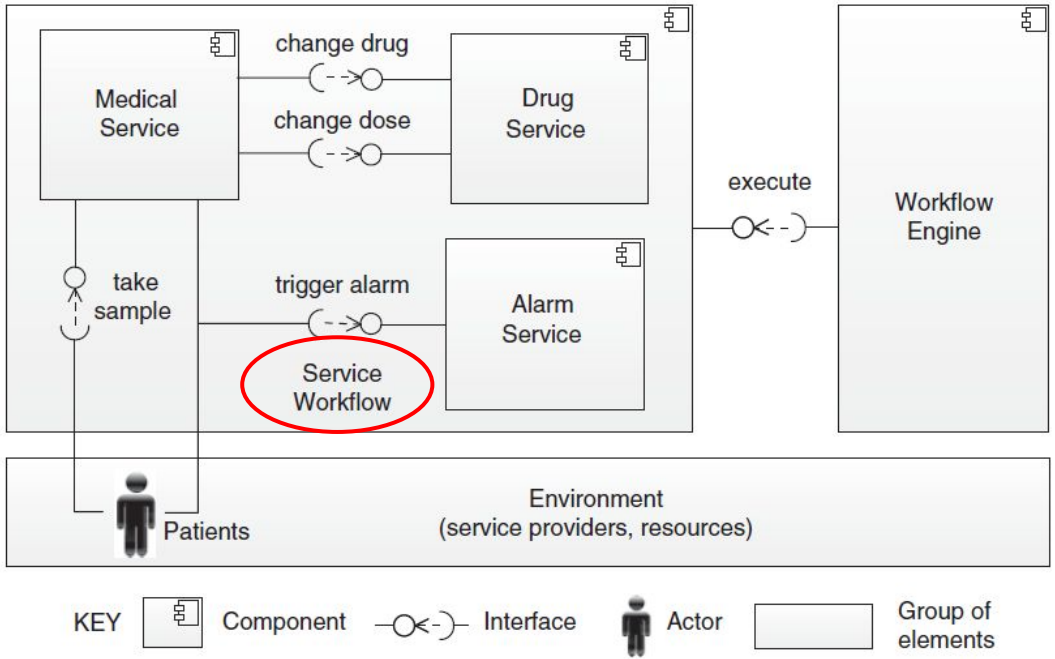


Managed System Example (1)

Adaptation:

1) allows the selection and use of concrete instances of the different types of services

2) provide support to **change service instances** in a consistent manner by ensuring that a service is only removed and replaced **when it is no longer involved** in any ongoing service invocation of the health assistance system



SAS Conceptual Model: Adaptation Goals

Represent concerns regarding the managed system adaptation goals: relate to quality properties of the managed system

Principal High-Level Types:

- **Self-configuration**: Systems automatically configuring themselves
- **Self-optimization**: Continually seeking ways to improve performance or reduce cost
- **Self-healing**: Detecting, diagnosing, and repairing problems
- **Self-protection**: Defending against attacks or cascading failures

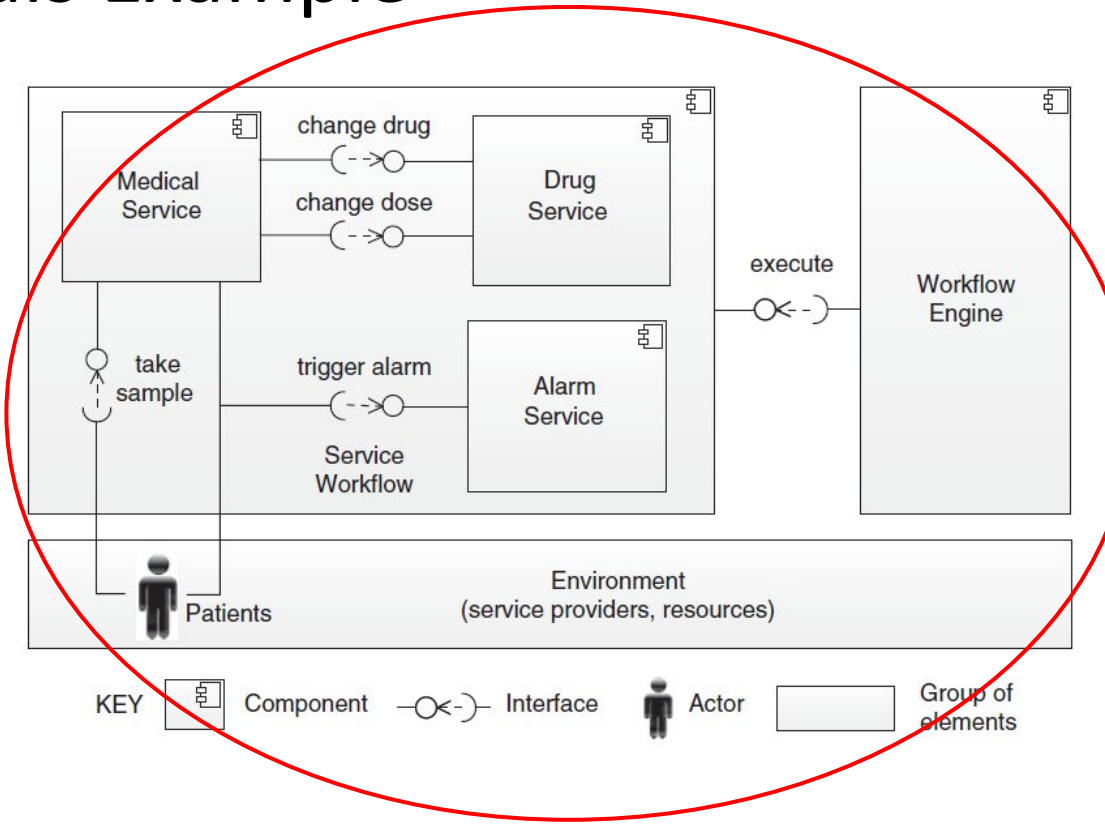
Adaptation goals can be subject to change, adding new goals or removing goals will require updates, and often also require updates of probes and effectors

Adaptation Goals Example

Adaptation goals:

1) the system **dynamically selects service instances** under changing conditions to keep the failure rate over a given period below a required threshold (self-healing goal), while the cost is minimized (optimization goal)

2) On the other hand, **adding a new adaptation goal**, for instance to keep the average response time of invocations of the assistance service below a required threshold, would be more invasive and would require an evolution of the adaptation goals and the managing system



SAS Conceptual Model: Feedback Loop (1)

Comprises the **adaptation logic** that deals with one or more adaptation goals

Feedback loop **monitors** the **environment** and the **managed system** and **adapts the latter** when necessary to realize the adaptation goals

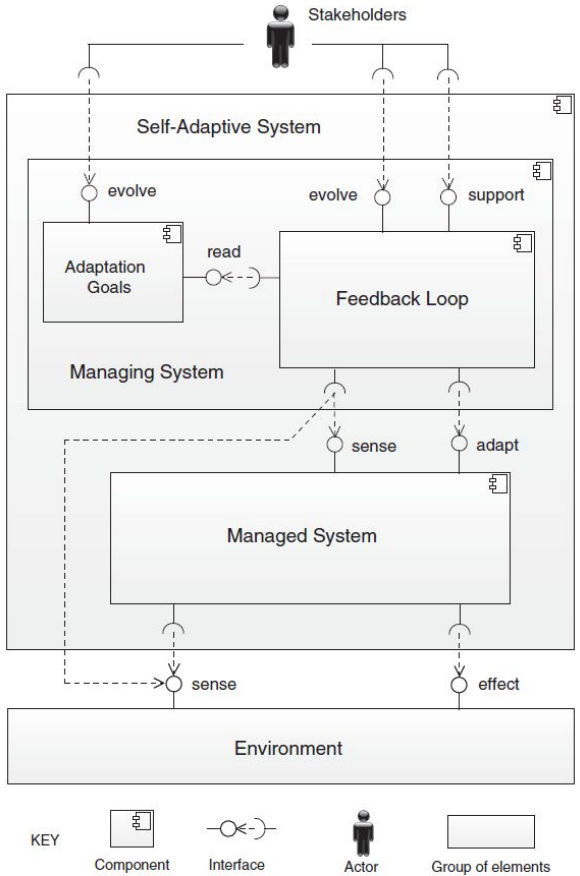
Reactive policy

The feedback loop **responds to a violation** of the adaptation goals by adapting the managed system to a new configuration that complies with the adaptation goals

Proactive policy

The feedback loop tracks the behavior of the managed system and adapts the system to **anticipate a possible violation** of the adaptation goals

SAS Conceptual Model



SAS Conceptual Model: Feedback Loop (2)

The managing system can be subject to change itself

Update a feedback loop to **resolve a problem** or a bug (e.g. add or replace some functionality)

To support **changing adaptation goals**, i.e. change or remove an existing goal or add a new goal

In both cases the need for evolving the feedback loop model is **triggered by users** based on observations

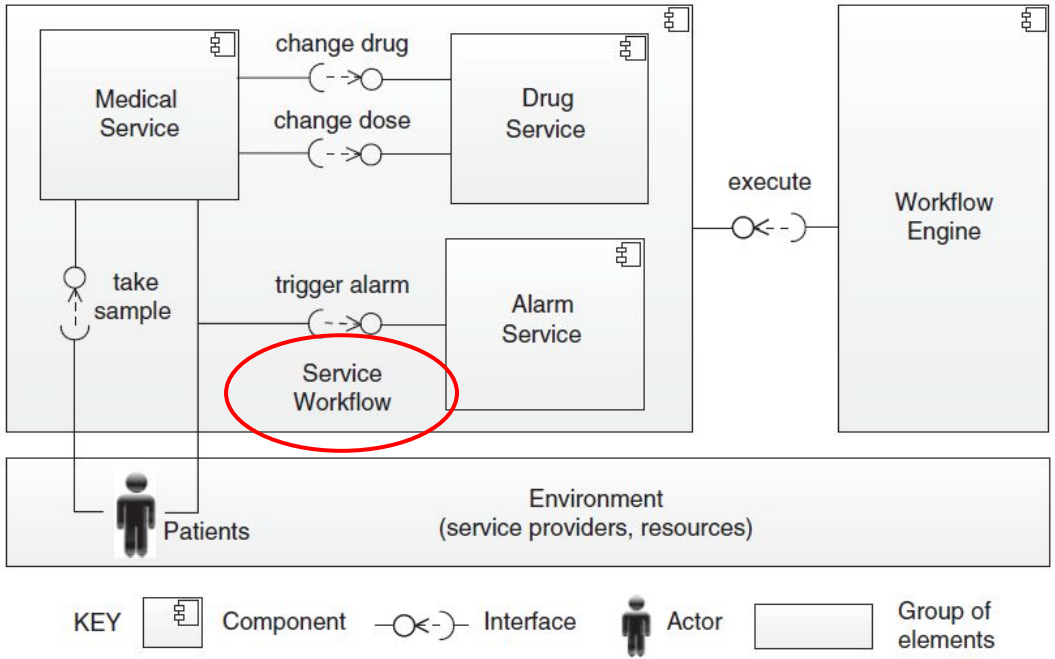
Feedback Loop Example

Feedback Loop:

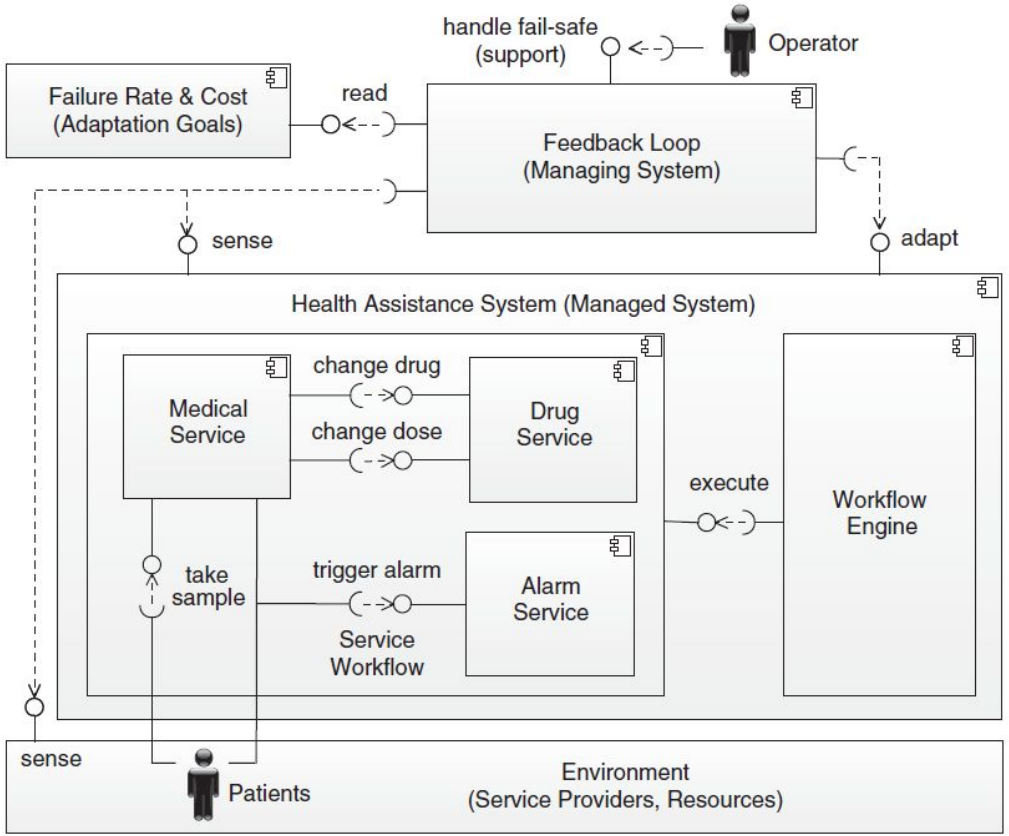
Ensures that the adaptation goals are realized monitoring the system behavior

1) **reactive policy**: the feedback loop will select alternative service instances

2) **proactive policy**: the managing system may involve a stakeholder to decide on the adaptation action to take



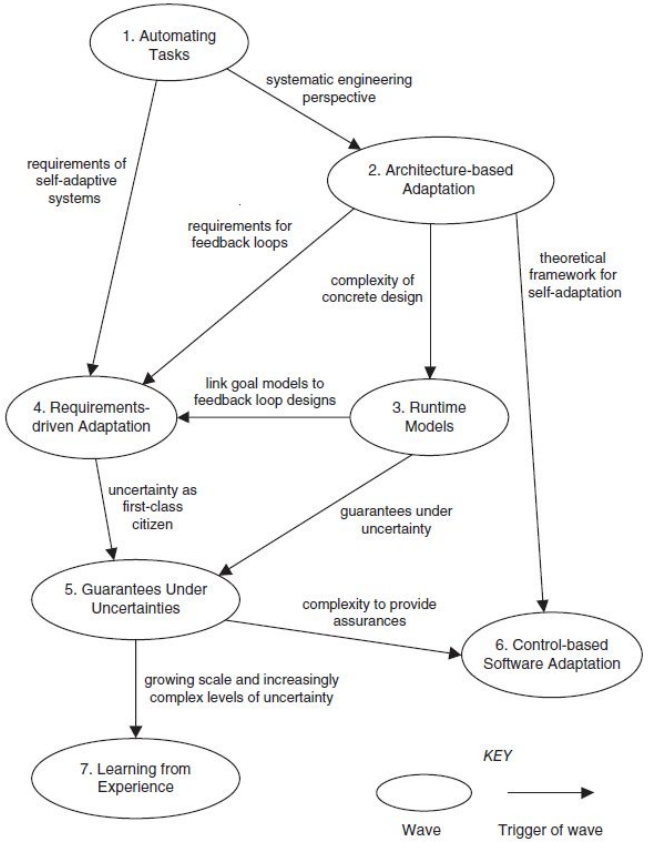
Conceptual model applied to a self-adaptive service-based health assistance system



Engineering Self-Adaptive Systems

The seven waves

The Seven Waves

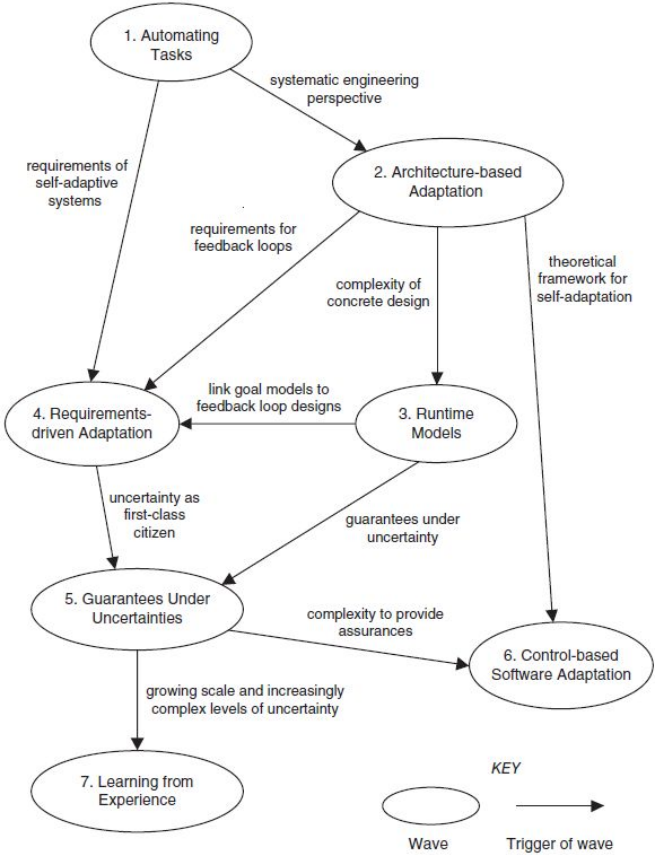


First Wave: Automating Tasks

Focuses on delegating complex and error-prone management **tasks from humans to the system**

A managing system (external manager) **monitors and adapts the system** automatically based on high-level objectives

The main goal is to automate management through **decomposition of essential management functions**

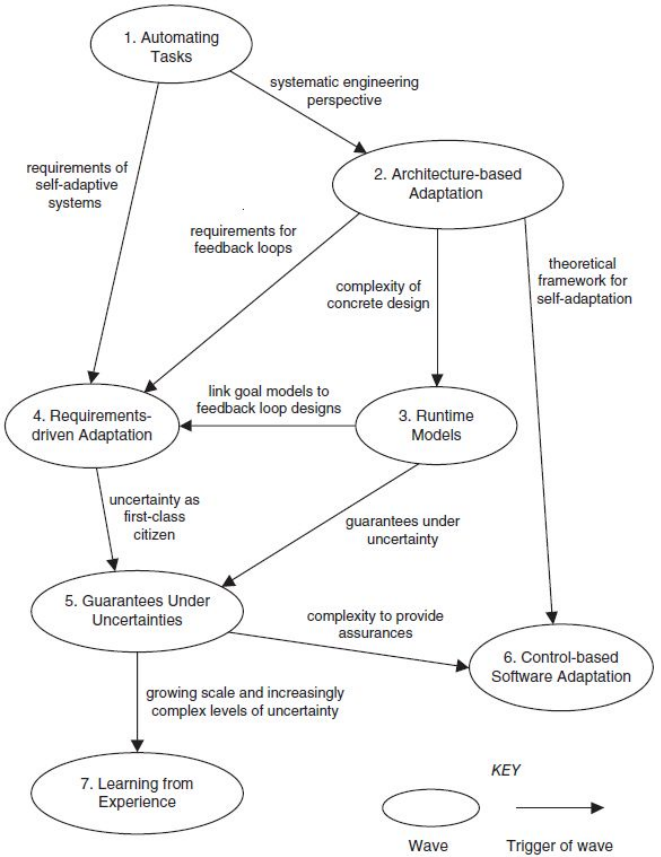


Second Wave: Architecture-based Adaptation

Introduces a systematic, engineering-oriented approach to self-adaptive systems using **software architecture principles**

Emphasizes abstraction and separation of concerns between **change management** (handling environmental changes) and **goal management** (handling changing objectives)

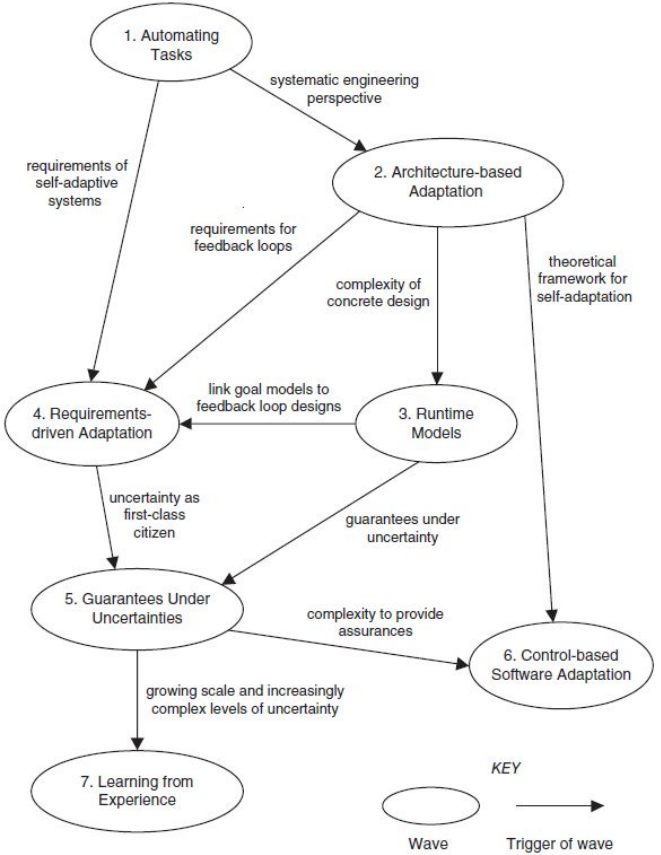
Uses **architectural models** to reason about system and environment changes effectively



Third Wave: Runtime Models

Addresses the complexity of implementing self-adaptive systems by **introducing runtime models** (models that exist and evolve while the system operates)

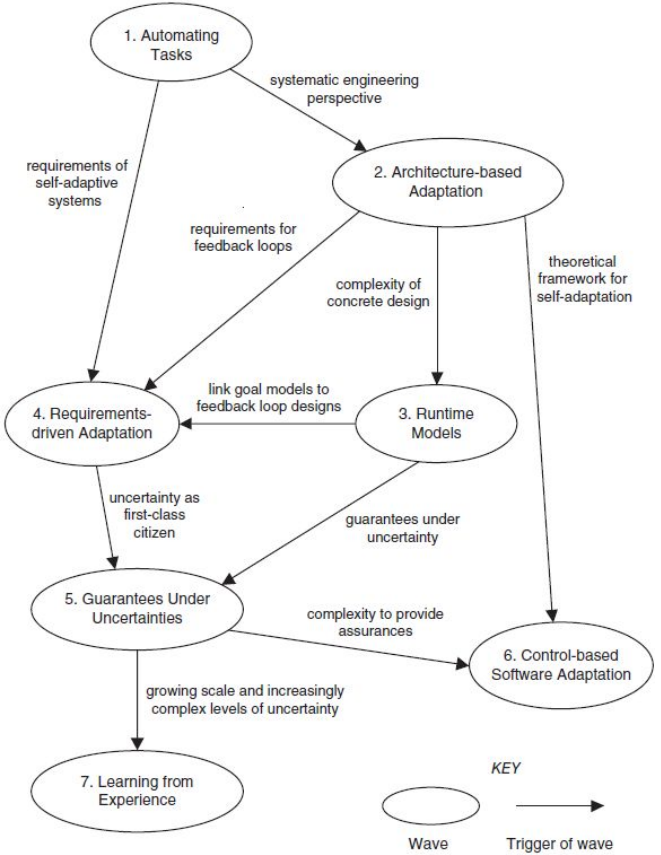
Extends model-driven development to the runtime, allowing systems to **analyze and make adaptation decisions during operation** using these live models



Fourth Wave: Requirements-driven Adaptation

Emphasizes **requirements as central elements** of self-adaptive systems, linking them to feedback loop design

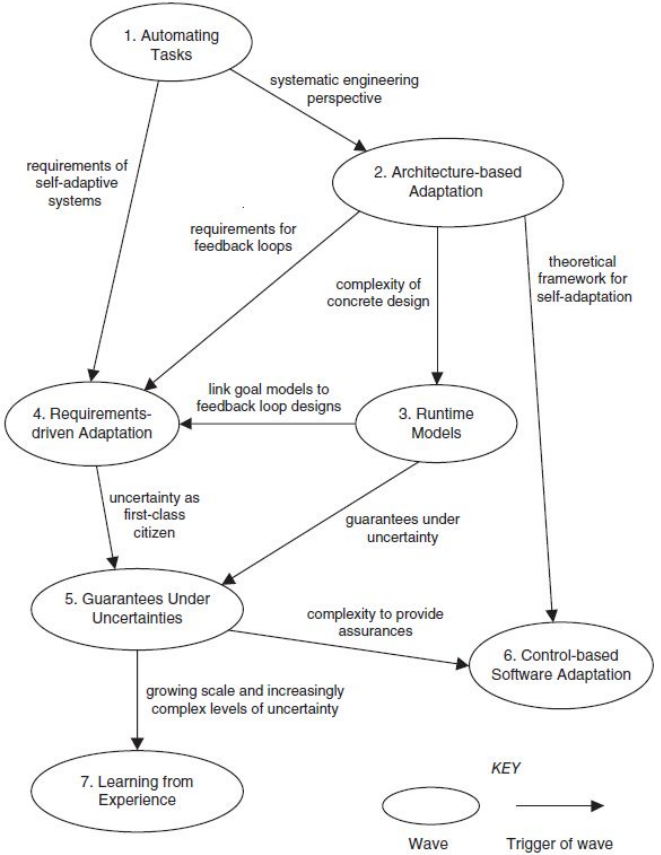
Distinguishes between requirements that define **adaptation goals** and those ensuring the **correct functioning** of the managing system itself.



Fifth Wave: Guarantees Under Uncertainty

Focuses on managing uncertainty in self-adaptive systems and **ensuring that adaptation goals are still met**

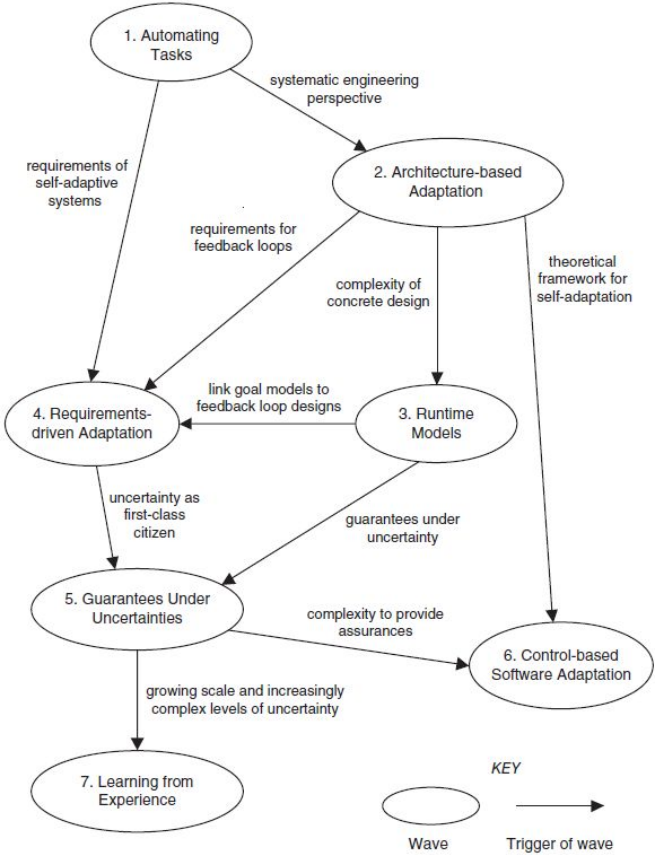
Aims to **provide trustworthiness** through the collection of evidence, both **offline** (pre-runtime) and **online** (during execution), demonstrating that goals are achieved despite uncertainty



Sixth Wave: Control-based Software Adaptation

Uses **control theory** to establish a formal, mathematical **foundation** for self-adaptive systems

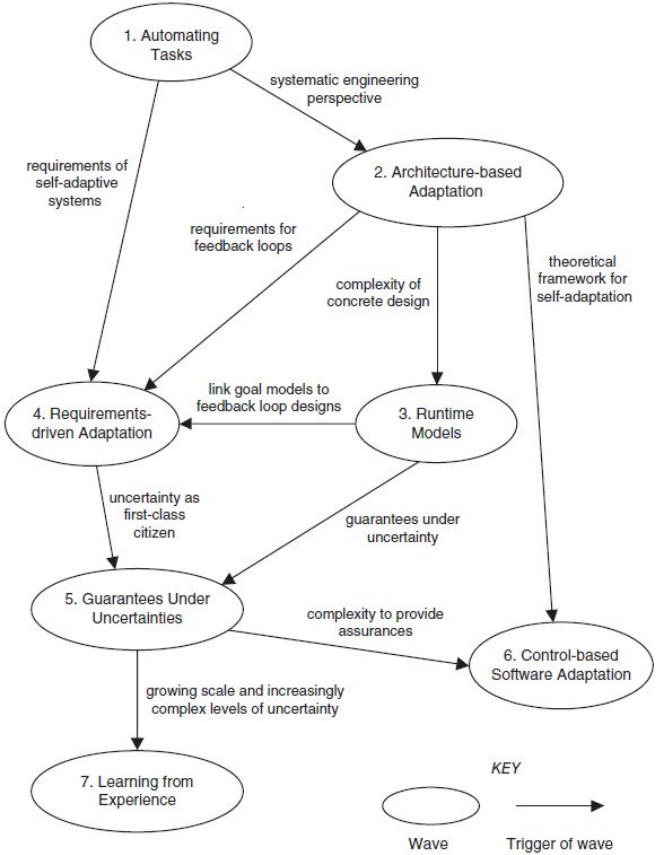
Involves defining adaptation goals, selecting controllers, and modeling the managed system to analyze and guarantee stability, performance, and other key properties



Seventh Wave: Learning from Experience

Leverages machine learning to enhance adaptation in large-scale and highly uncertain systems

Learning techniques support updating runtime models, reducing adaptation search spaces, predicting adaptation outcomes, and improving decision-making efficiency



Wave I

Automating Tasks

Motivation

In the late 1990s–2000s, installing and maintaining software systems required extensive manual work, leading to serious management challenges

IBM introduced the concept of **autonomic computing**, aiming to create systems that manage and adapt themselves based on high-level goals set by administrators

At the core is a feedback loop that collects data and acts automatically to maintain objectives, reducing manual effort and improving system efficiency and responsiveness

Maintenance Task: Self-Optimization

Self-optimization is the capability of a system to **continuously improve** the use of its limited resources such as memory, computational power, bandwidth, or energy while maintaining the required quality levels. This property ensures system sustainability and business profitability by avoiding resource waste.

Maintenance Task: Self-Healing

Complex systems often face software **bugs or hardware faults that can disrupt normal functioning**. Manually identifying and fixing such issues is time-consuming and costly

Refers to a system's ability to **detect, diagnose, and recover from failures automatically**, ensuring continuous and reliable operation

A self-healing system continuously monitors itself to detect anomalies, localizes the source of the problem, and takes corrective actions to restore normal operation

Maintenance Task: Self Protection

While tools like firewalls and intrusion detection systems provide basic defense, **complex systems often remain vulnerable and difficult for humans to monitor** effectively

Capability of a system to **defend itself against malicious attacks and to anticipate potential threats** that could compromise its operation

A self-protecting system can detect suspicious behavior, take preventive or corrective actions, and contain cascading effects caused by security breaches

Maintenance Task: Self-Configuration

In large-scale systems, manual **installation and configuration of new elements is complex**, time-consuming, and error-prone

Capability of a system to **automatically integrate new components or elements** according to high-level objectives, without disrupting normal operation

Self-configuration ensures that new elements are seamlessly added, enabling the system to adapt and scale efficiently while maintaining continuous operation

Reference Model

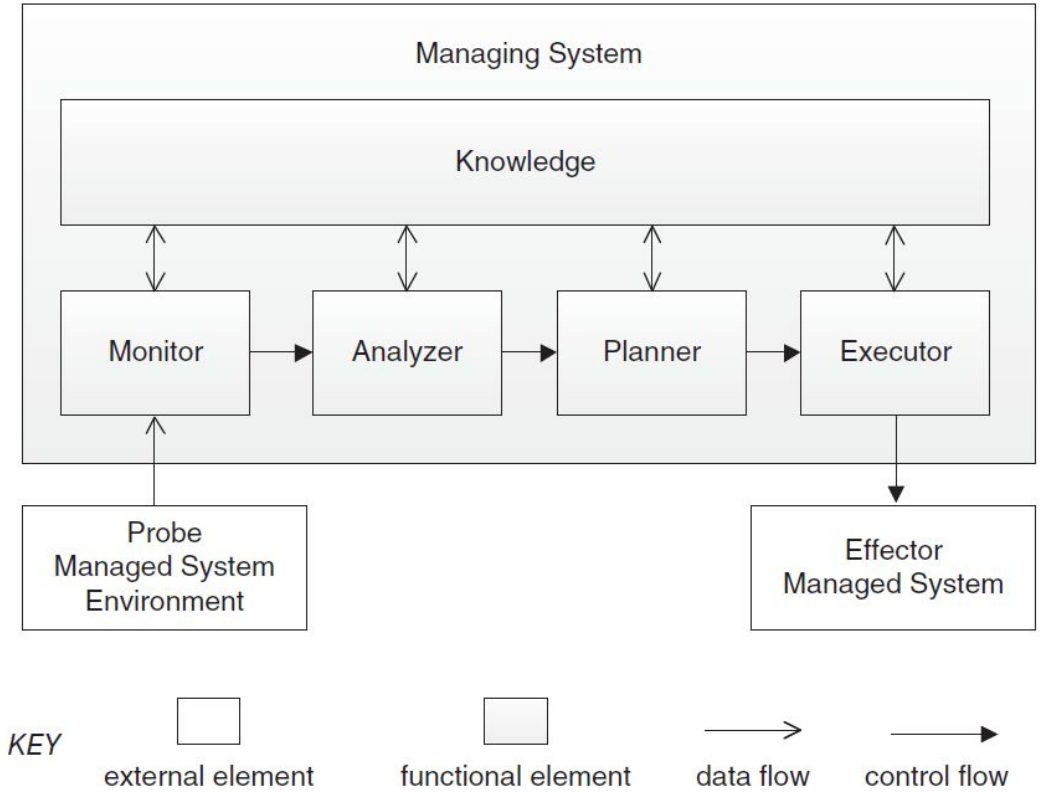
Key Characteristics:

1) **Functional decomposition:** Breaks down self-adaptation into core functions

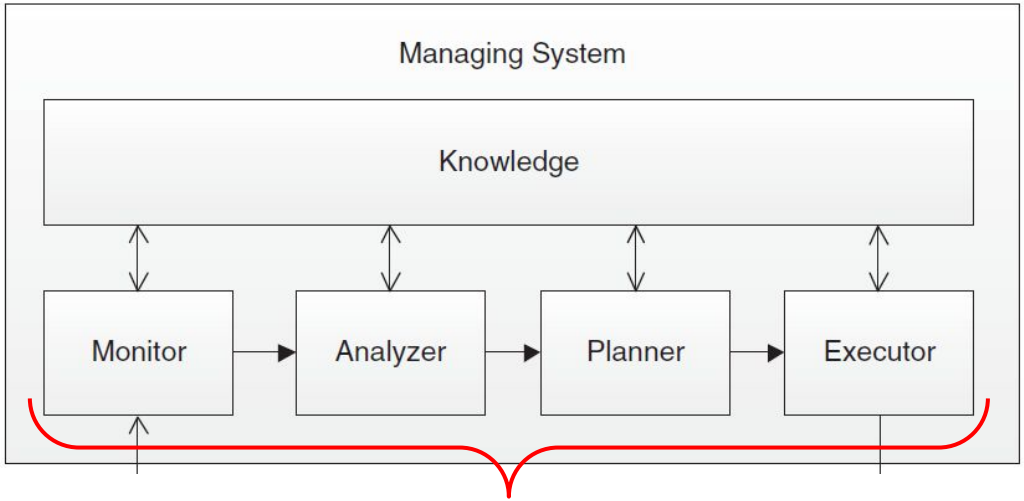
2) **Cooperation:** Defines how elements interact and share information

3) **Experience-based:** Consolidates domain knowledge and maturity of self-adaptive systems

4) **Independent of implementation:** Focuses on **what** needs to be done, not **how** it is mapped to software



MAPE-K



Realize the four basic functions of any self-adaptive system sharing common Knowledge

Hence the reference model of a managing system is often referred to as the **MAPE-K model**

MAPE-K Elements

Monitor:

- Collects data from the system and environment via probes
- Updates the Knowledge base with processed data

Analyzer:

- Uses up-to-date knowledge to detect the need for adaptation
- Analyzes possible adaptation options

Planner:

- Selects the best adaptation option
- Creates a plan with one or more actions to move the system from its current to desired configuration

Executor:

Implements the plan via effectors, adapting the system as required

Knowledge Element

Provides the **shared knowledge and mechanisms** for MAPE elements to work together through **reading, writing, notification and updating** functionalities

Ensures **consistency** when multiple clients access knowledge simultaneously (avoiding race conditions)

Four types of knowledge

Managed System Model

A managed system model represents the system that is managed by the managing system

- Focuses on parts relevant to **self-adaptation**
- Enables **self-awareness**

The environment model represents elements or properties of the environment in which the managed system operates

- Represents **relevant aspects of the system's environment**
- Enables **context-awareness**

Adaptation Goals Model

The model of the adaptation goals maintains a representation of the objectives of the managing system

Can be expressed as:

- **Rules** with constraints and priorities
- **Fuzzy goals** with intermediate truth values (0–1)
- **Utility functions** representing preferences

Enables **goal-awareness**

MAPE Working Models

A MAPEworking model represents knowledge that is shared between two or more MAPE elements

Often **domain-specific**, e.g., quality prediction or adaptation plans.

Parameterized models allow predictions for different system configurations.

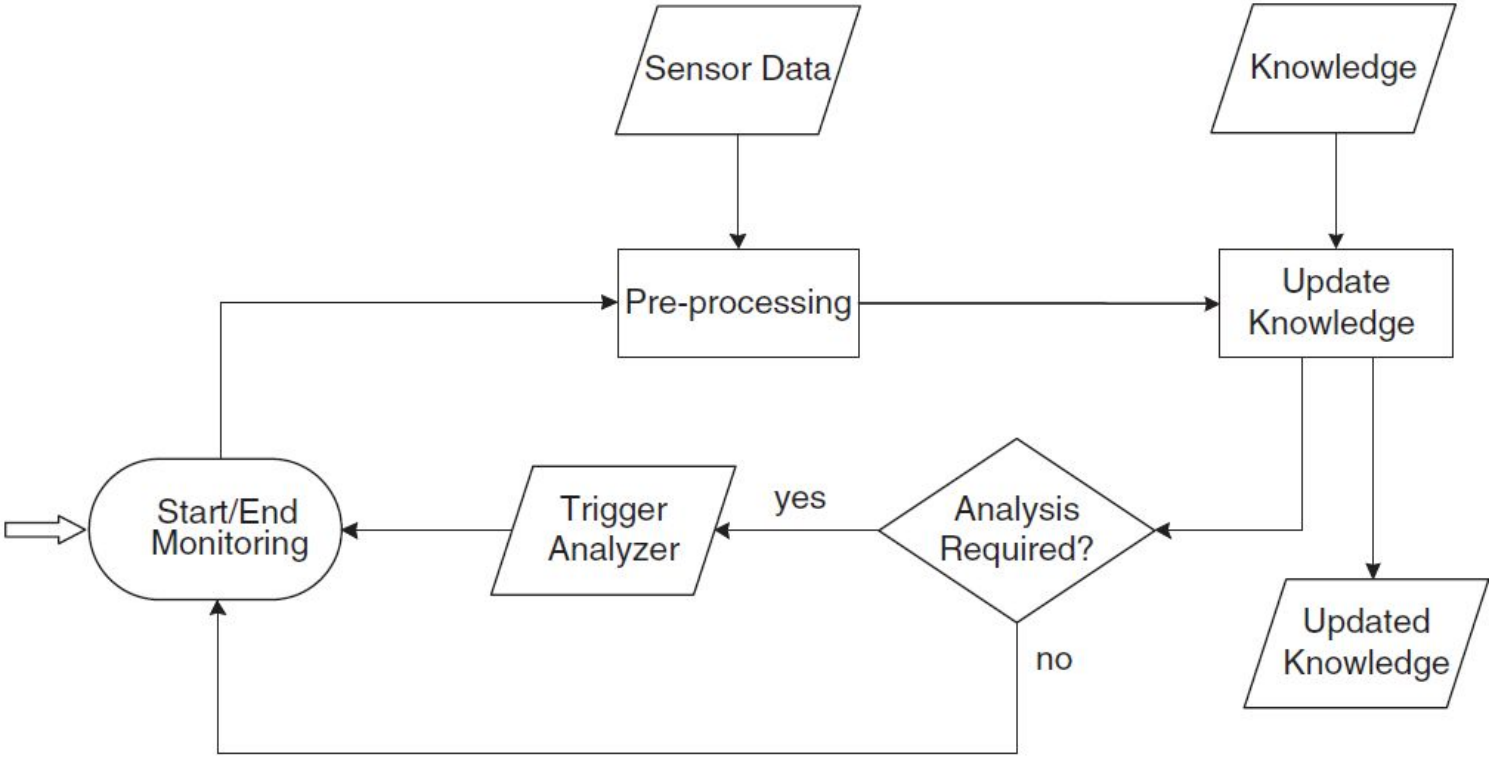
Monitor Function

Monitor Function has a dual role

Keeps **track of the environment** in which the managed system operates and **reflects the relevant changes** in the environment model

Keeps track of **changes of the managed system** and **updates the managed system model** accordingly

Monitor Workflow

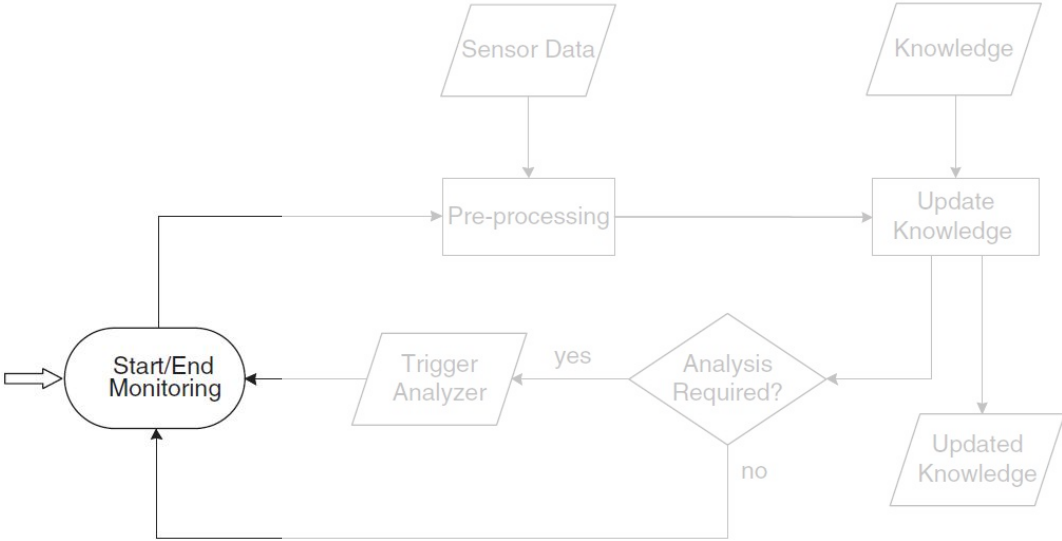


Monitor Workflow: Activation

External trigger: e.g., probe finishes a sensing cycle.

Periodic trigger: based on predefined time windows.

Continuous cycle: automatically restarts after each monitoring cycle.

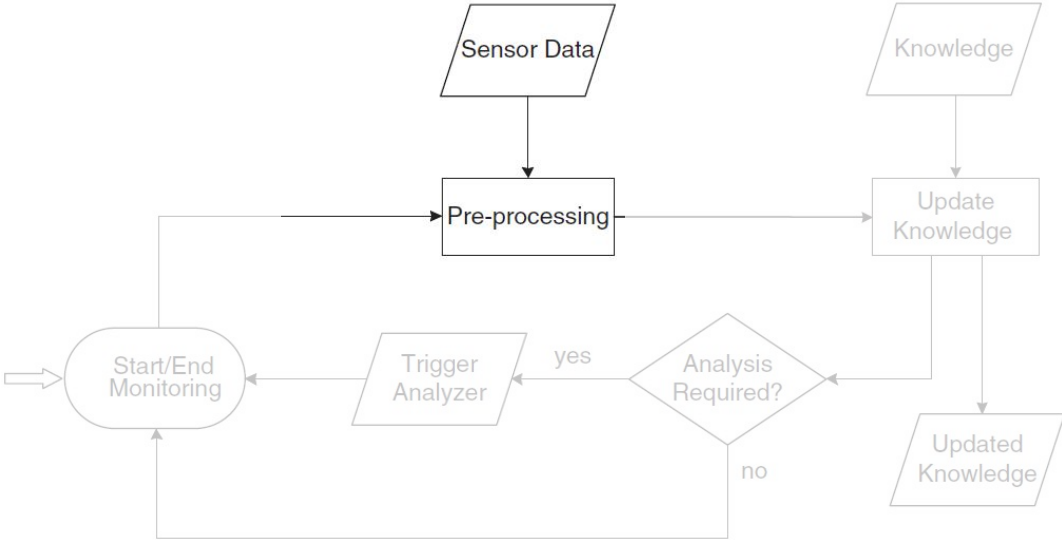


Monitor Workflow: Data Collection

After activation, the monitor **collects data** taken from sensors

Data **may require some form of pre-processing** before it can be used to update the knowledge models

- **Simple**: filtering, aggregation
- **Advanced**: Bayesian estimation, deep neural networks

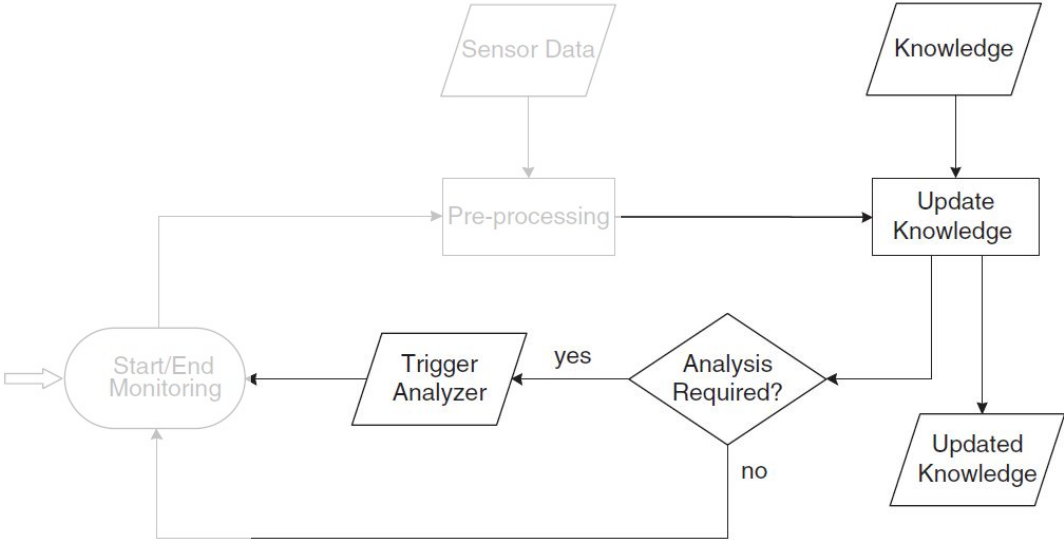


Monitor Workflow: Updating Knowledge

The monitor uses the current knowledge and the pre-processed data to **update the knowledge models**

The monitor may perform a check to identify **whether an analysis step is required or not**

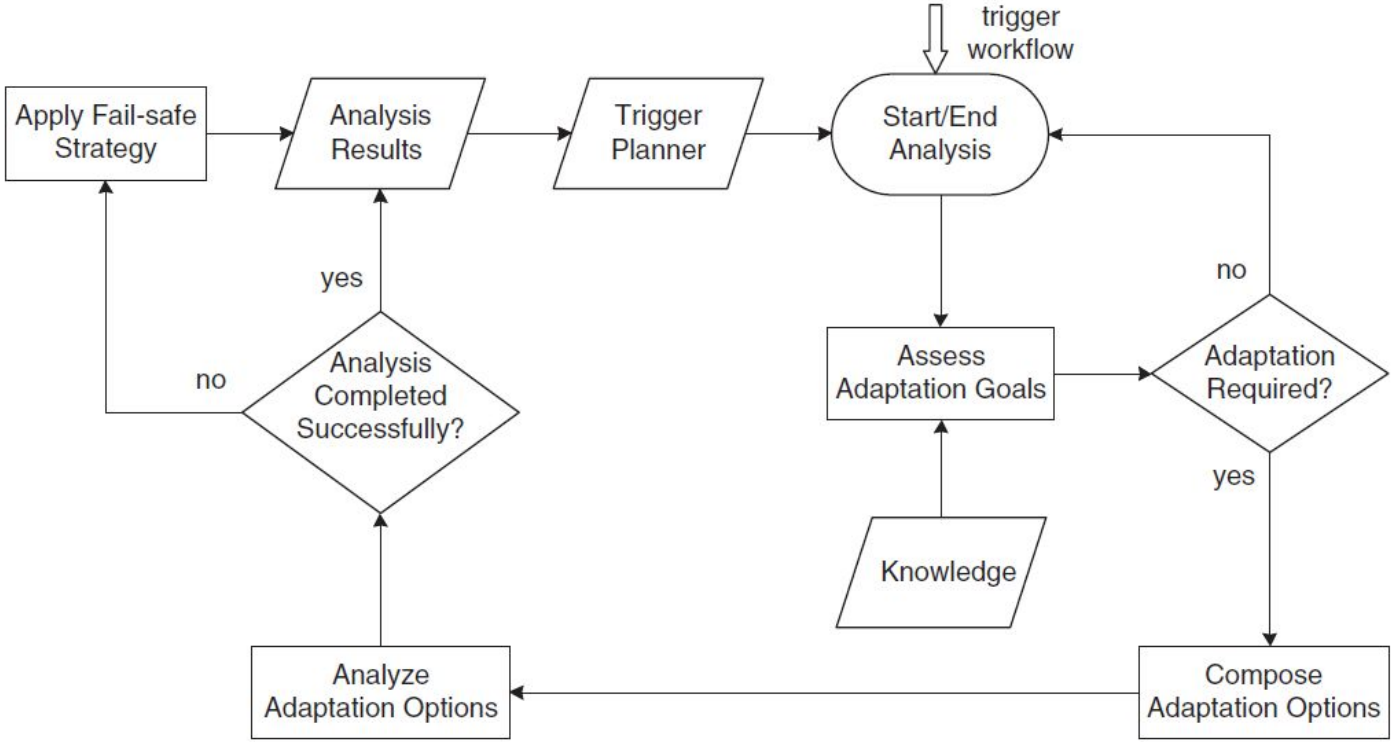
Simple check: values exceed thresholds
Advanced check: detect trends or patterns



Analyzer Function

The role of the analyzer function is to **(1) assess** the up-to-date knowledge and **(2) determine** whether the system satisfies the adaptation goals or not, and if not, to **(3) analyze** possible configurations for adaptation

Analyzer Workflow

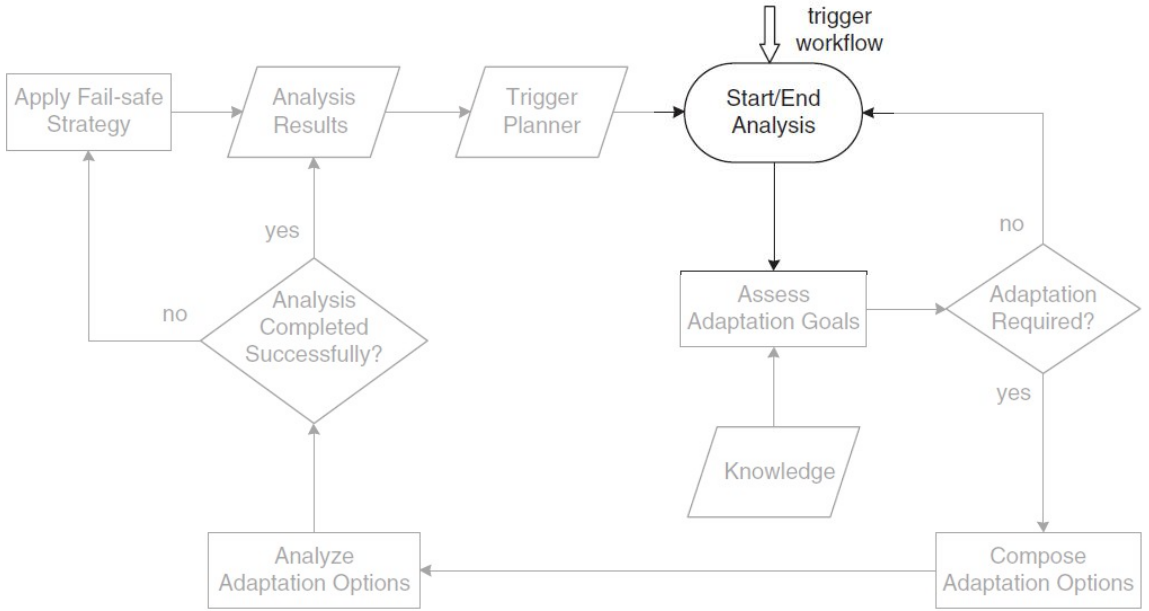


Analyzer Workflow: Activation

Externally triggered: e.g., by Monitor.

Periodic trigger: fixed time intervals.

Continuous cycles: restarts after previous cycle.

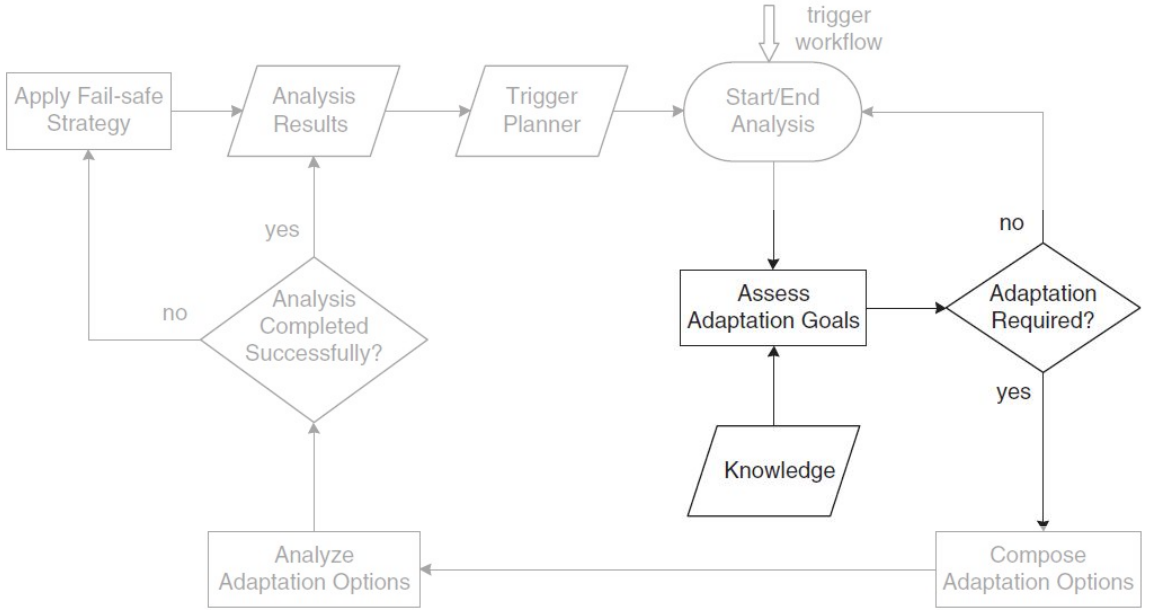


Analyzer Workflow: Assessment

Assesses the actual conditions based on the knowledge (adaptation goals) to determine whether adaptation needs or not

Simple: Check if any adaptation goal is violated → initiate adaptation

Advanced: Compute utility combining weighted quality properties

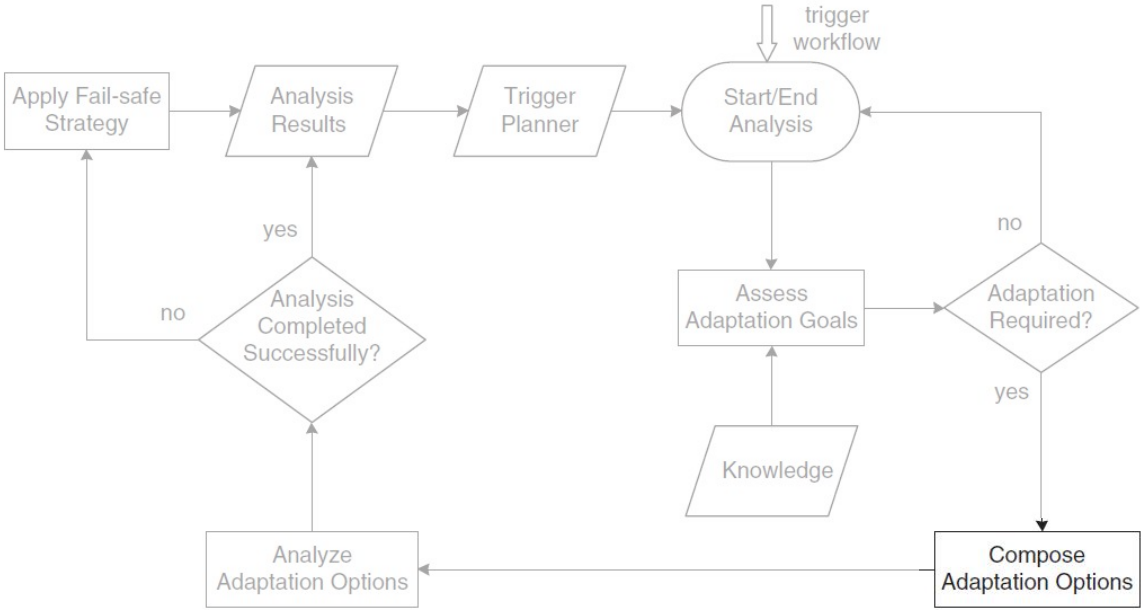


Analyzer Workflow: Adaptation (1)

Adaptation options are composed

Set of configurations reachable from the current configuration by adapting the managed system

Exhaustive, Distance-based, Advanced

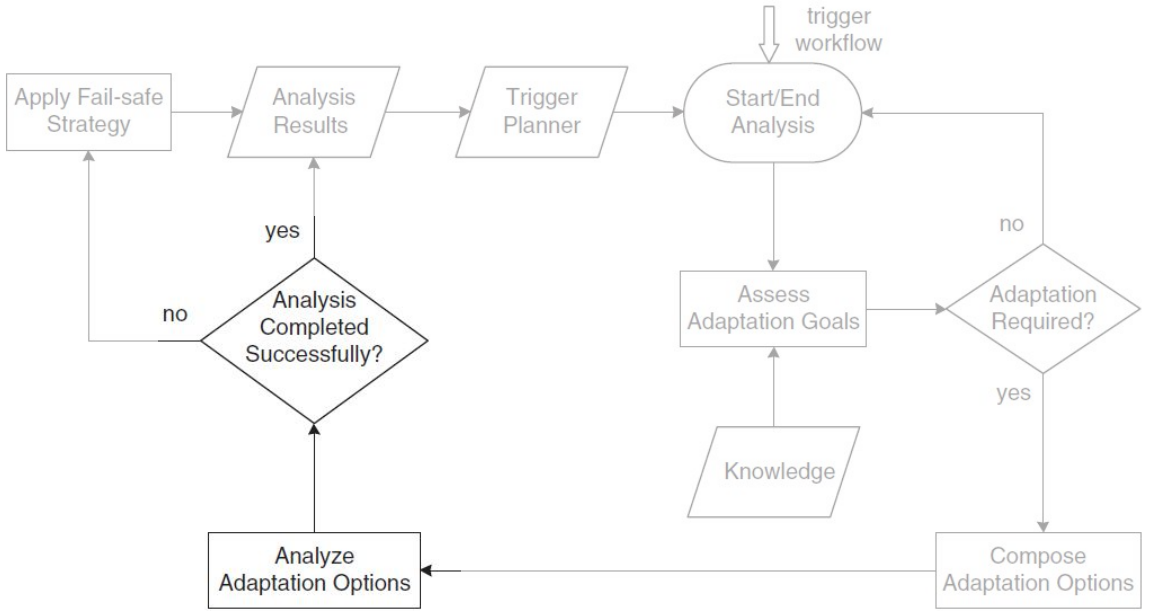


Analyzer Workflow: Adaptation (2)

Defines how well each option meets **adaptation goals**

Mechanism types

- **Reactive:** uses past knowledge
- **Active:** uses current system state
- **Proactive:** predicts future outcomes

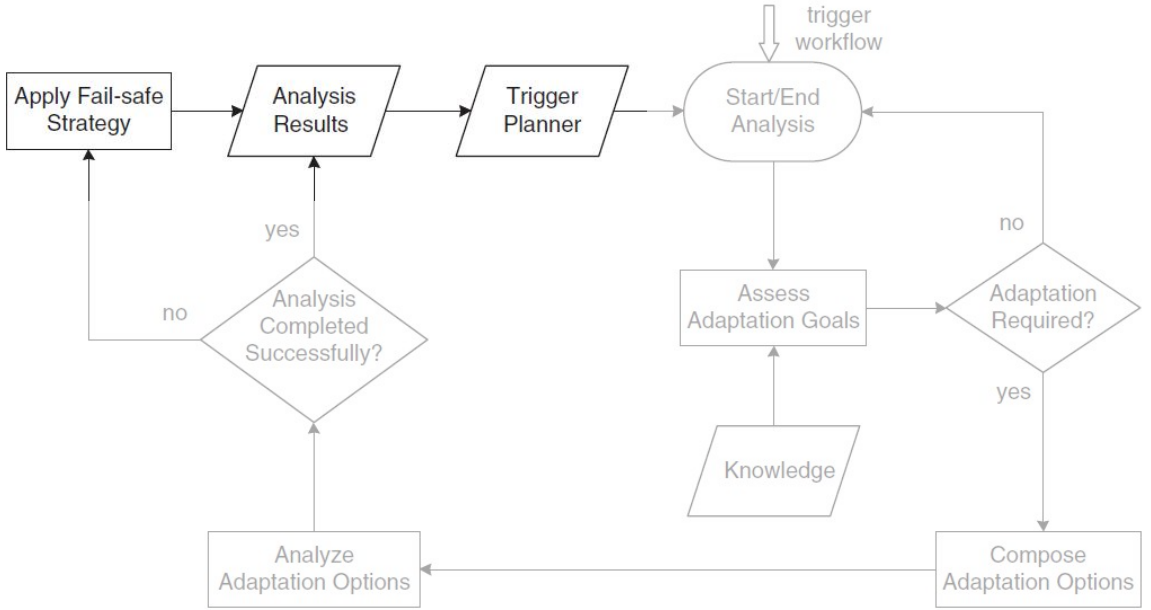


Analyzer Workflow: Fail-Safe Strategies

When preparing the adaptation plan, analysis may fail

Alternative choices:

1) Do not adapt the system
2) Apply predefined safe configuration
3) Bring system to safe stop

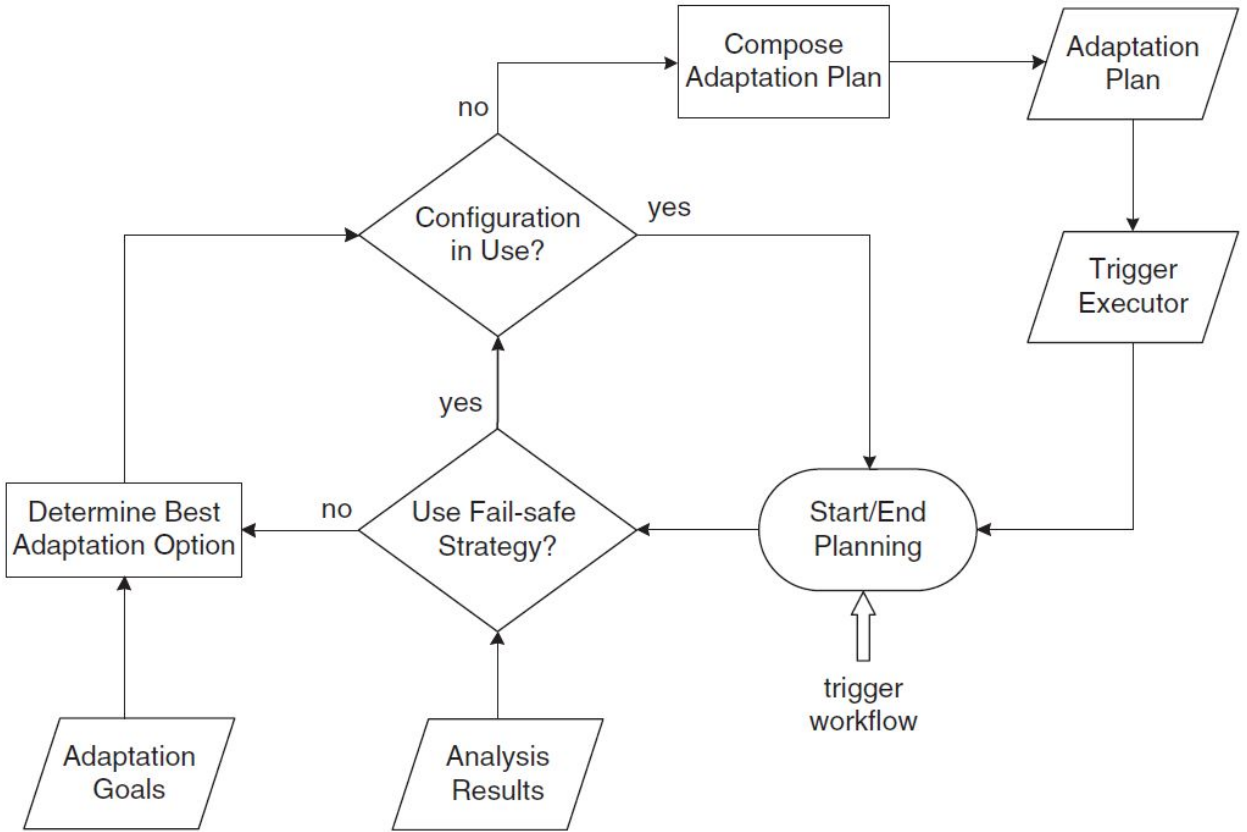


Planner Function

The role of the planner function is to s**elect the best adaptation option and generate a plan for adapting the managed system** from its current configuration to the new configuration defined by the best adaptation option

In the event that a fail-safe strategy needs to be applied, the planner only needs to **generate an adaptation plan to bring the system to a safe state**

Planner Workflow

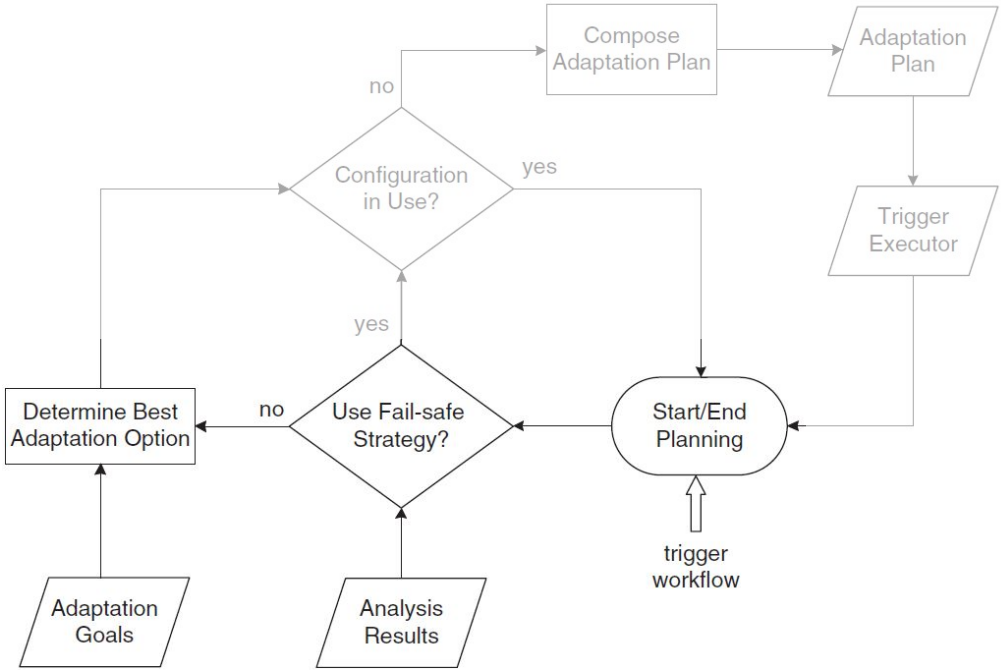


Planner Workflow: Activation

the planner **determines the best adaptation option** based on the adaptation goals

Two mechanisms for selecting:

1. **Rule-based goals**
2. **Utility-based goals**



Planner Workflow: Activation

the planner determines the best adaptation option based on the adaptation goals

Two mechanisms for selecting:

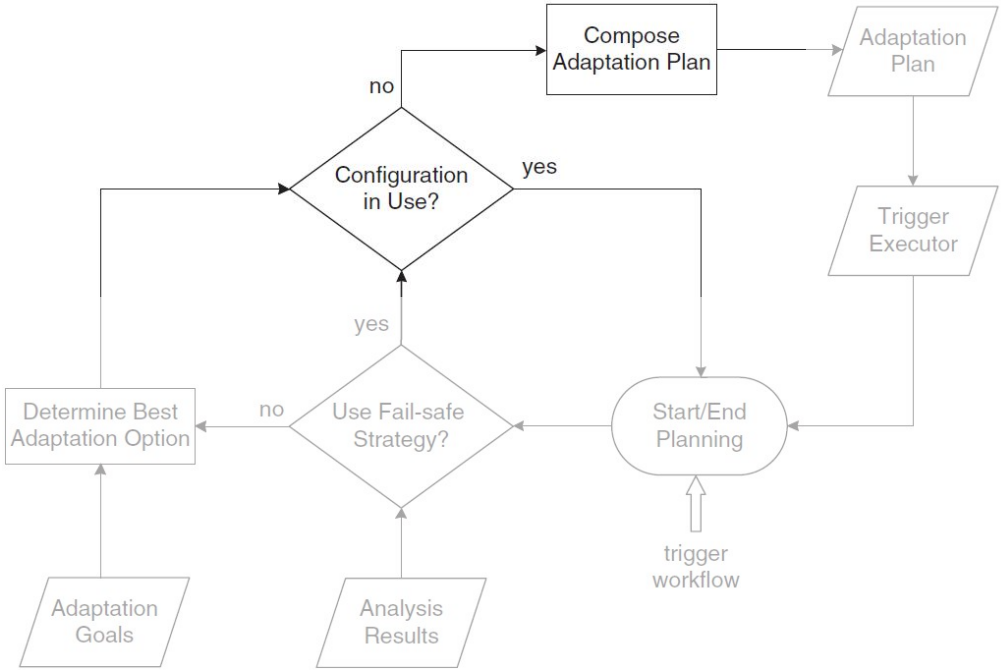
1. **Rule-based goals**
   a. Divide options into compliant vs. non-compliant with threshold rules
   b. Rank compliant options based on minimizing or maximizing quality properties
2. **Utility-based goals**
   a. Rank options based on expected accumulated utility

Planner Workflow: Planning and Adaptation Actions

The Planner creates a **plan** (sequence of actions) detailing the **actions** (executable units) needed to adapt the system

Planning criteria:

- **Quality:** likelihood of achieving adaptation goals
- **Timeliness:** speed of generating the plan
- Tradeoff



Planning Mechanisms

Several planning mechanisms exist. The choice depends on system complexity and required performance

Reactive planners:

- Fast, use **condition-action rules** or **finite state machines**
- Parameterized for the current context

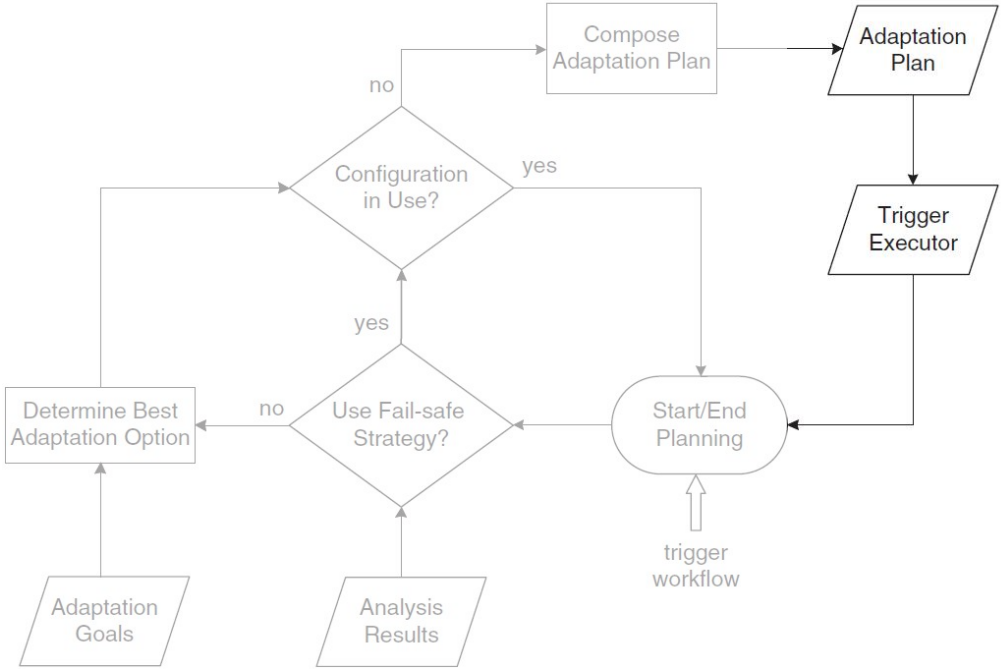
State-variable planners:

- Represent **system states and actions**
- May face **combinatorial explosion** in large systems

After plan generation, the Planner **triggers the Executor function**

Planner Workflow: Planning and Adaptation Actions

Once the adaptation plan has been generated, the planner triggers the executor function



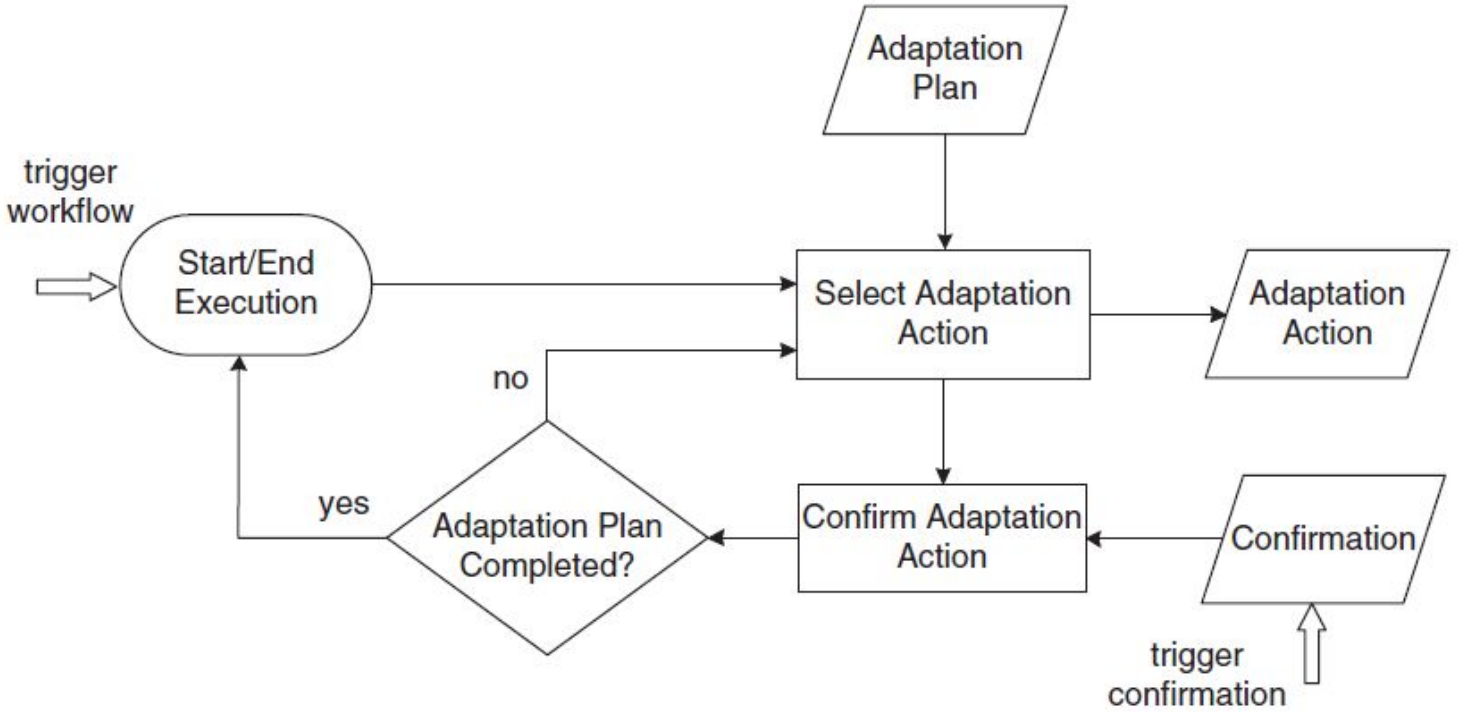
Executor Function

The Executor function is responsible for **carrying out the adaptation plan** generated by the Planner. It **applies adaptation actions** to the managed system, ensuring that the system transitions from its current configuration to the new configuration.

Key Points:

- Activated externally by the Planner
- Executes the **adaptation plan** step by step
- Uses **effectors** to enact changes on the managed system
- Waits for **confirmation** that each action has taken effect before proceeding

Executor Workflow

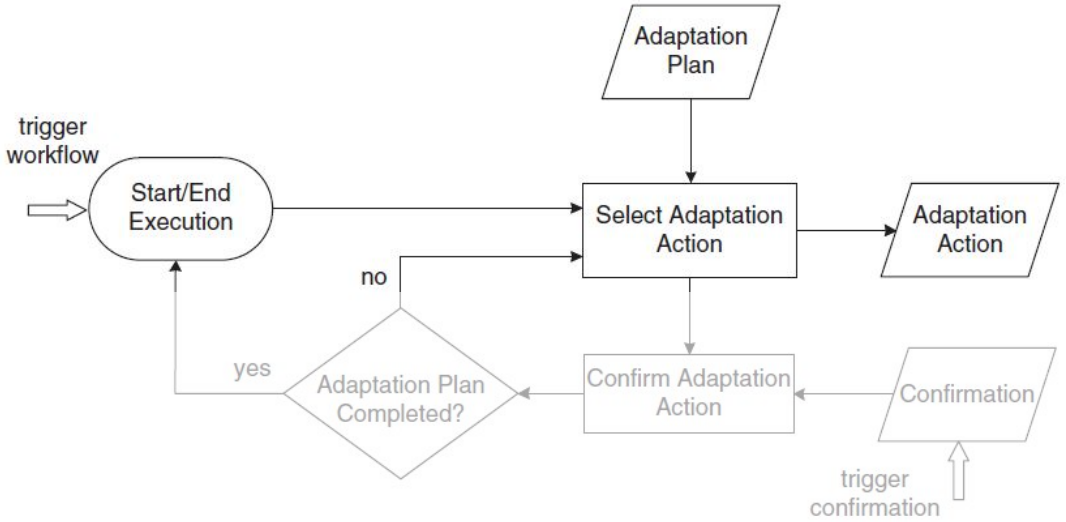


Executor Workflow: Executing Actions

Adaptation actions are executed according to the plan (ordered or unordered)

Ordered plans: first action executed first, following the sequence

Unordered plans: any action can be selected

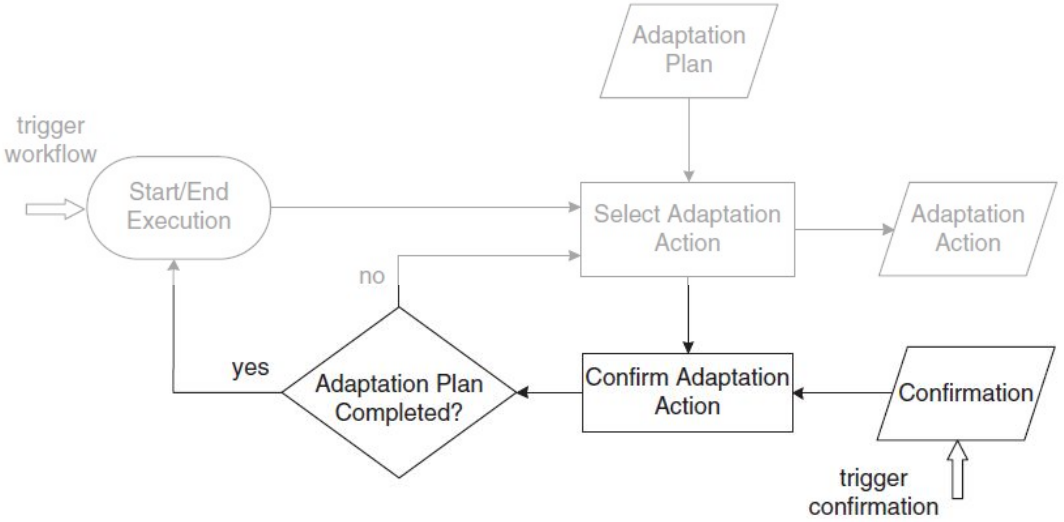


Executor Workflow: Confirmation and Plan Completion

After executing an adaptation action, the Executor **confirms its effect** and decides whether to continue with the next action or end the plan

Confirmation can be obtained via:

- **Direct notification** from the managed system
- Delegation to the **Monitor** function
- Waiting for **predefined time windows**

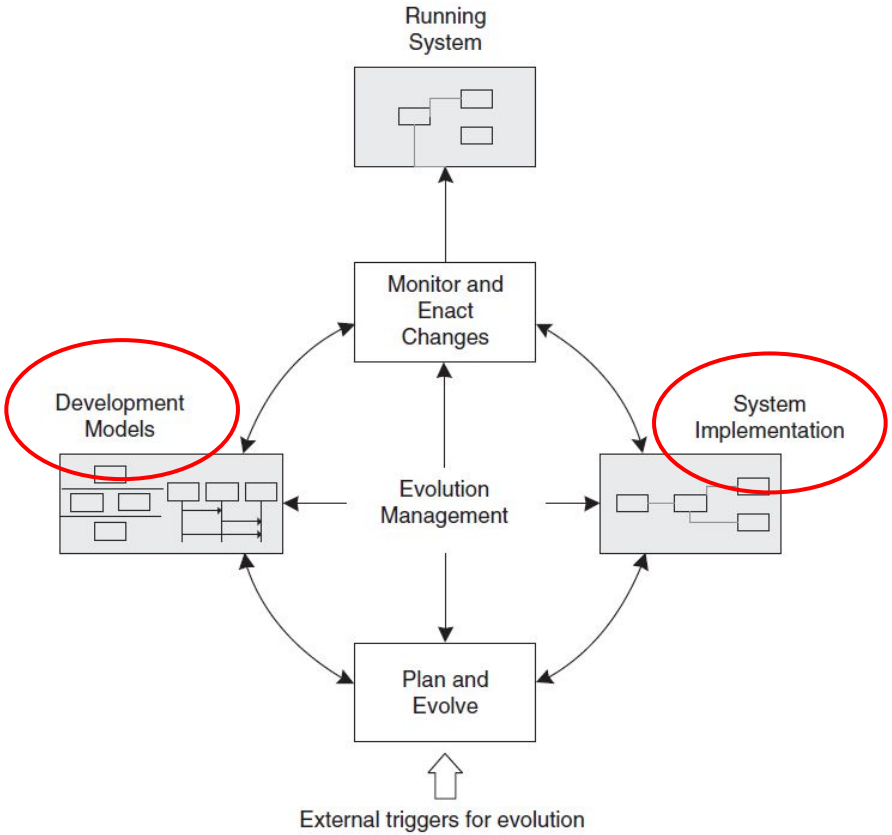


Software Evolution Overview

Software evolution is the process of **repeatedly updating software after its initial deployment**. Updates are necessary to correct faults, improve performance, adapt to environmental changes, or add new functionality. Modern systems demand updates with minimal or no service interruption.

Evolution Management Artifacts and Activities

Development models can be very diverse, including specifications of requirements, design, deployment, processes, etc



The system implementation includes the actual code, supporting infrastructure, deployment scripts, and other related artifacts

Triggers for Software Evolution

Software evolution can be triggered externally, **based on planned changes**, or internally, in **response to system observations**

- **External triggers:** requests for new functionality, strategic release plans
- **Internal triggers:** detection of bugs, failure to meet performance or quality objectives

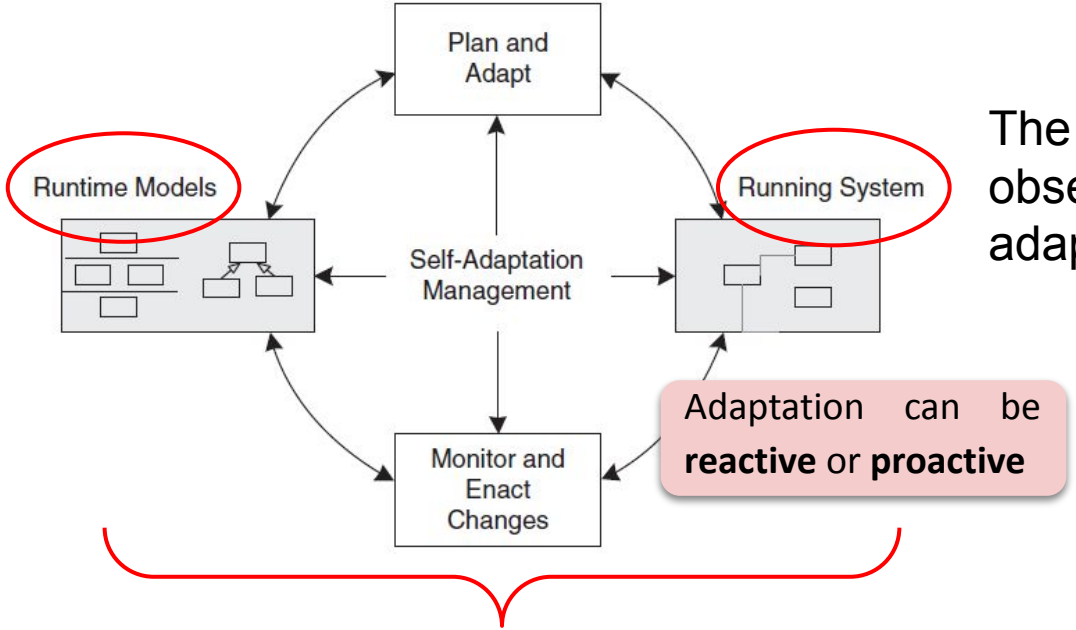
Enables **proactive and reactive evolution**

Self-Adaptation Management Overview

Self-adaptation management is the process through which a software system **automatically adjusts its behavior or structure in response to internal or external changes**. It ensures that the system continues to meet its goals without requiring constant human intervention.

Core Artifacts and Activities

Represent system state, environment, and goals (Knowledge)



The system being observed and adapted

Activities (MAPE loop)

Integrating Software Evolution and Self-Adaptation (1)

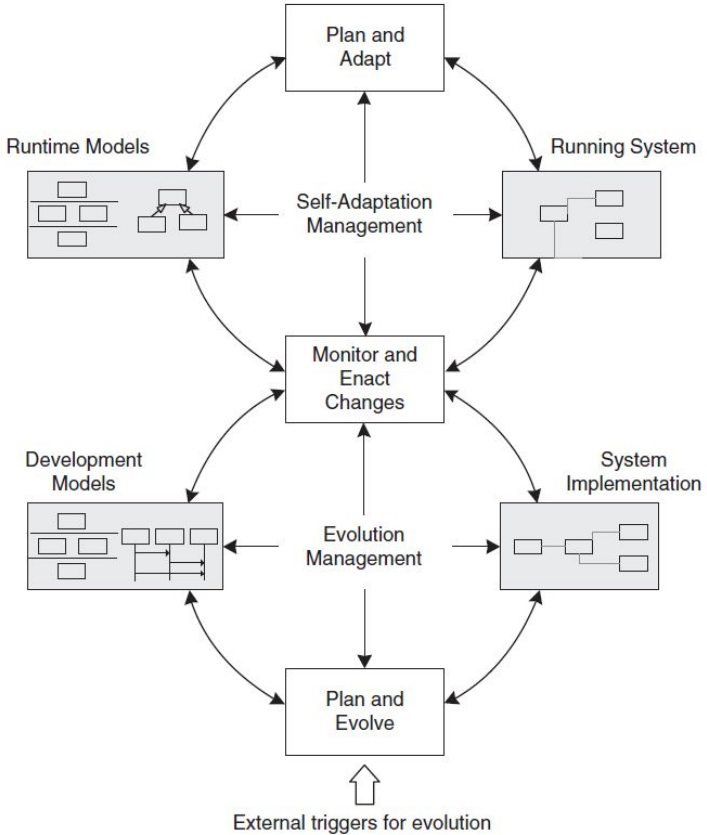
Evolution management and **self-adaptation management** are **complementary** activities that deal with different types of change

Handles ***unanticipated*** change; requires human involvement

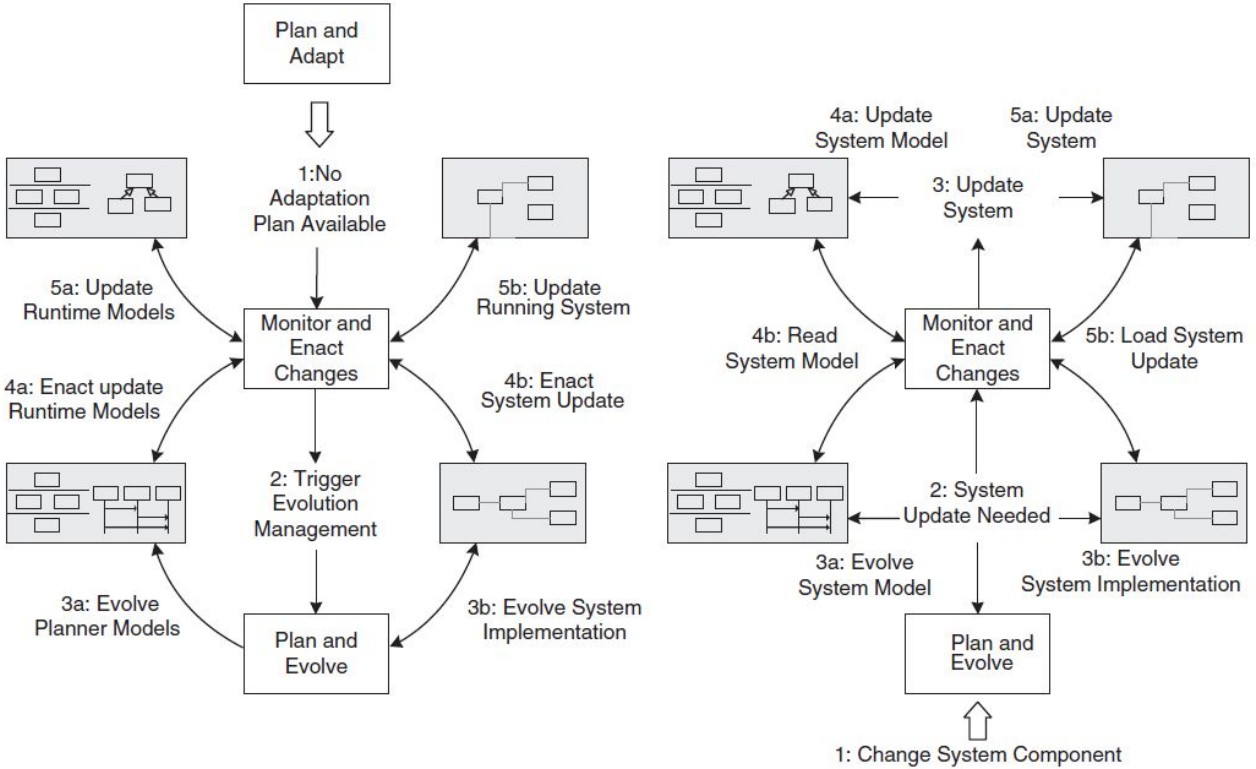
Handles ***anticipated*** change; may operate automatically

Anticipated = the system has been built such that it can detect the change and handle it in some way

Integrating Software Evolution and Self-Adaptation (2)

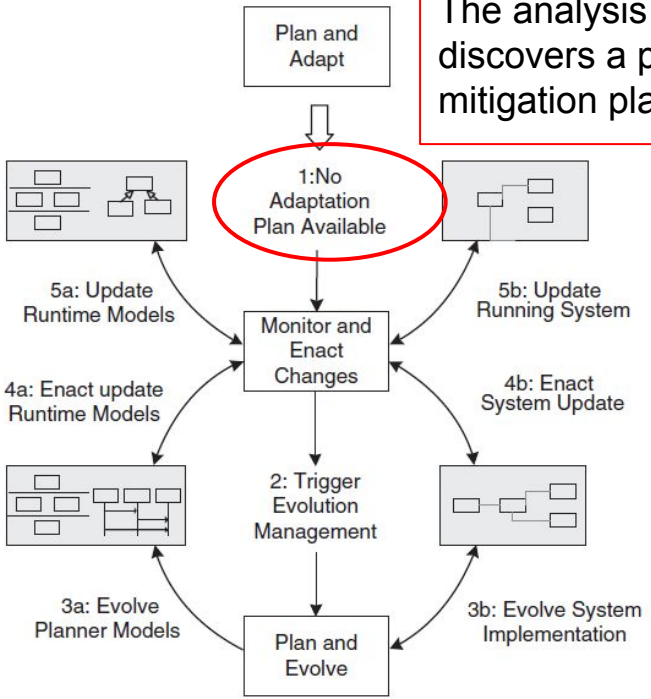


Interacting activities

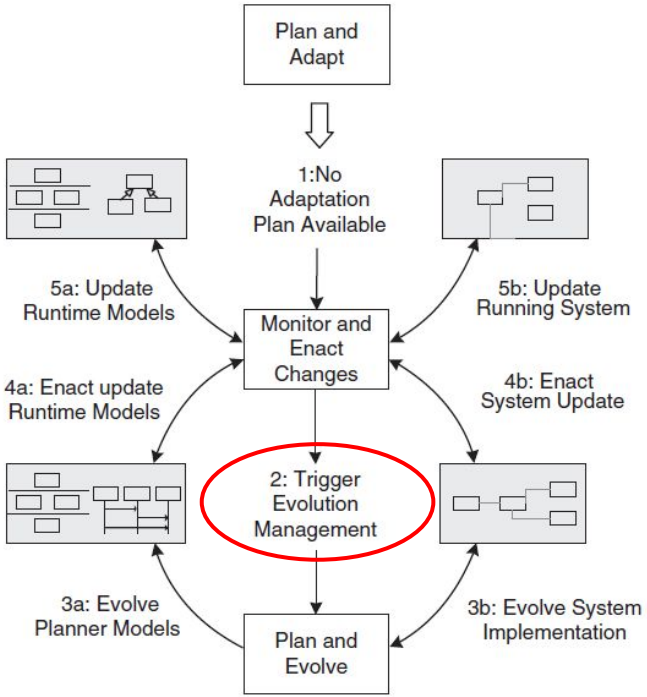


Adaptation Triggers Evolution (1)

The analysis element of the feedback loop discovers a problem for which no mitigation plan is available

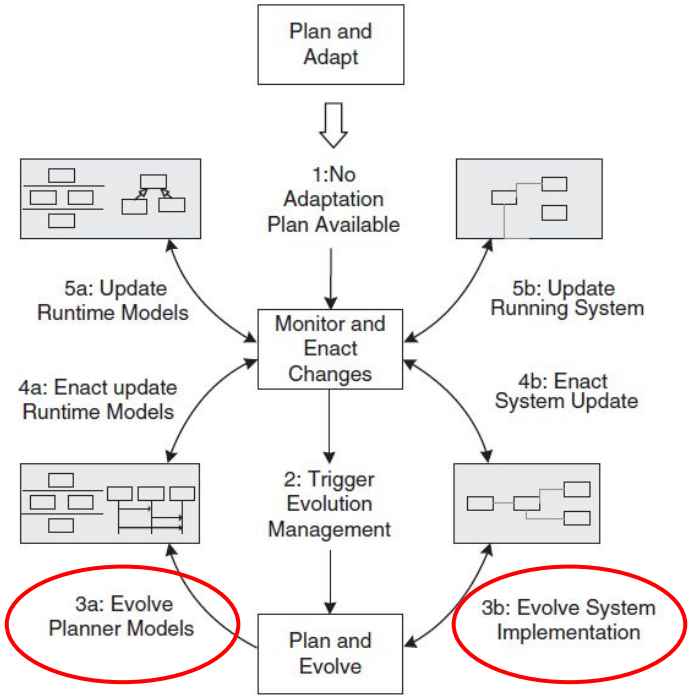


Adaptation Triggers Evolution (2)



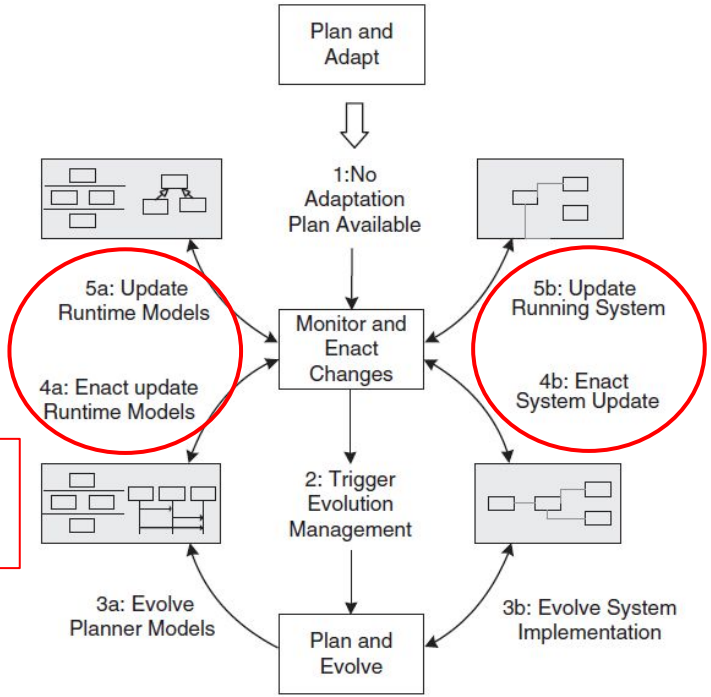
This triggers evolution management, which will process the request

Adaptation Triggers Evolution (3)



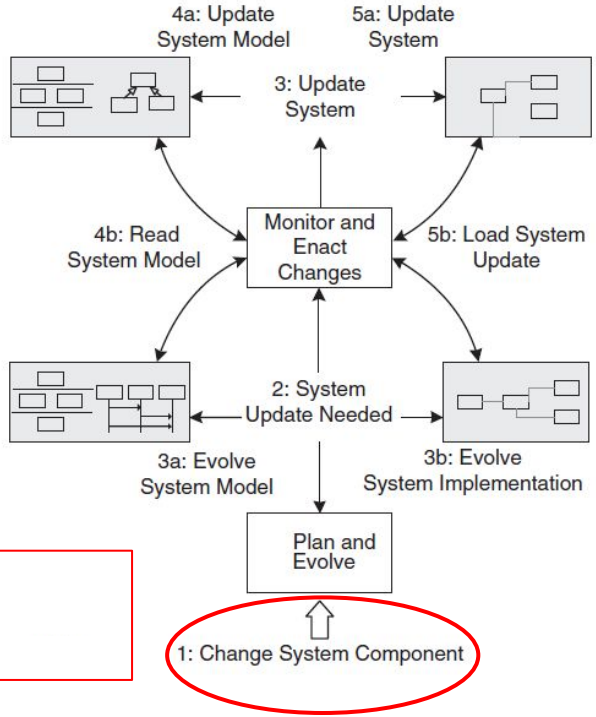
The evolved planner models and corresponding implementations will be added

Adaptation Triggers Evolution (4)



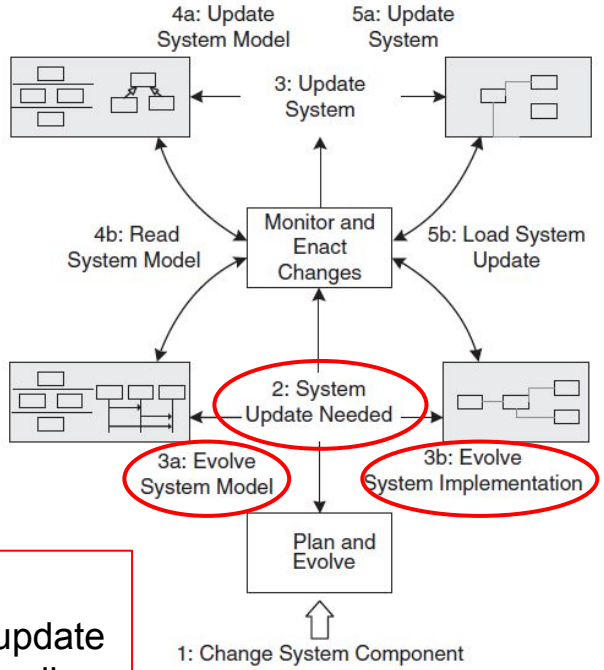
Finally, the running system will be updated, resolving the initial problem

Evolution Triggers Adaptation (1)



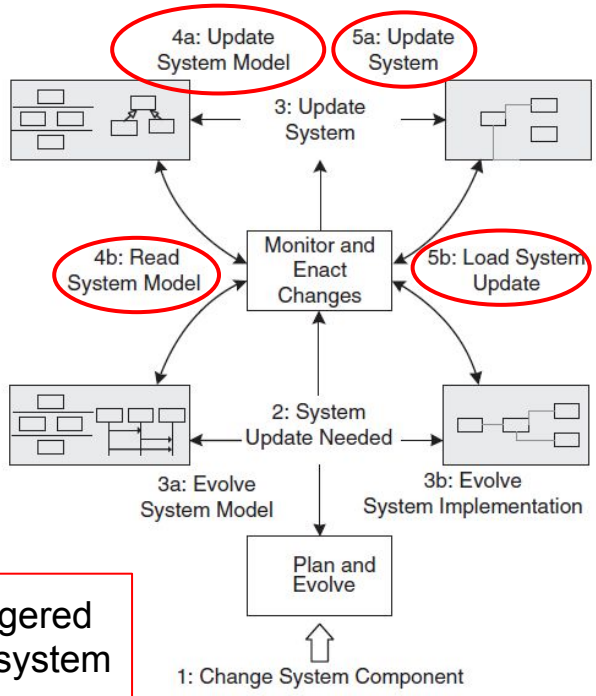
An update of the running managed system is requested

Evolution Triggers Adaptation (2)



This request will trigger evolution management, which will initiate an update of the system model and a corresponding evolution of the system implementation

Evolution Triggers Adaptation (3)



Adaptation management will be triggered to update the runtime model of the system and the running code accordingly

Wave II

Architecture-based Adaptation

Engineering SAS Architecture

Wave I ⟶ motivation, set of principles and concepts

Wave II ⟶ abstractions that enable designers to define self-adaptive systems

⟶ modeling abstractions that enable the system to reason about change

Handling Change in Software Systems

Changes were usually managed using **internal mechanisms** tied directly to the system's code (e.g., exc protocols, etc.)

Tightly coupled with code → difficult to modify and reuse

However, modern approaches like self-adaptation use **external mechanisms**

Enable modularity, reusability, and easier updates

Why a Software Architecture Perspective?

A software architecture perspective offers a structured way to manage runtime adaptation using external feedback loops ensuring maintainability and scalability

1. *Separation of concerns*
2. *Integrated approach*
3. *Leveraging consolidated efforts*
4. *Abstraction to manage system change*
5. *Dealing with system-wide concerns*
6. *Facilitating scalability*

Why a Software Architecture Perspective?

Separation of concerns: divide a software system into parts that address specific goals. Separating domain functionality, handled by the managed system, from adaptation logic, handled by the managing system

Integrated approach: connects all stages of system engineering, from configuration and deployment to runtime adaptation and maintenance

Leveraging consolidated efforts: By relying on proven specification languages, design patterns, and tactics, engineers can reuse consolidated knowledge and methods

Abstraction to manage system change: Modeling a system as components and their connections allows engineers to reason about adaptation at a higher level, focusing only on what matters for change while hiding unnecessary detail

Dealing with system-wide concerns: An architectural view enables a global understanding of the system. It allows monitoring and reasoning about system-wide properties such as performance, reliability, and security

Facilitating scalability: support composition and hierarchy, making it possible to manage systems at multiple levels of granularity. This property is essential for the self-adaptation of large-scale and complex applications

Three-Layer Model for Self-Adaptive Systems

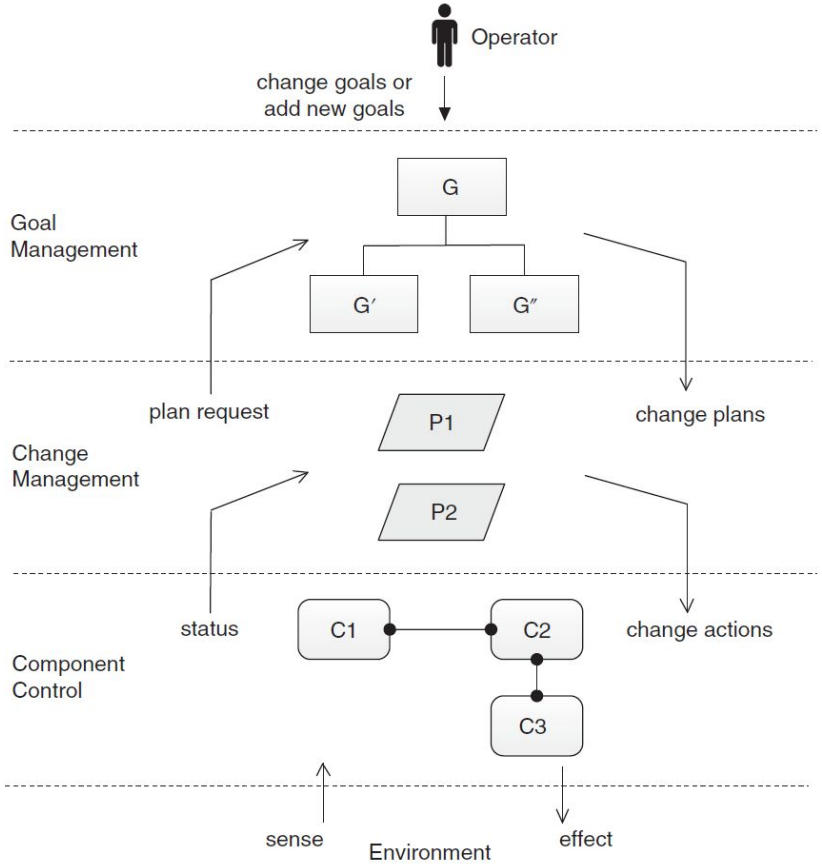
The three-layer model provides an architectural view of self-adaptive systems, organizing adaptation activities into distinct layers of increasing complexity

Reactive **Planning** **Strategy**

Inspired by robotic architectures

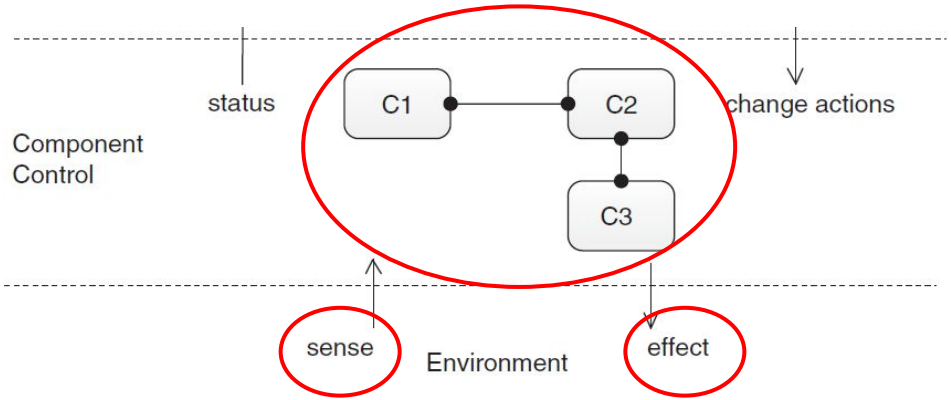
Three-Layer Model

realizes a MAPE-based workflow



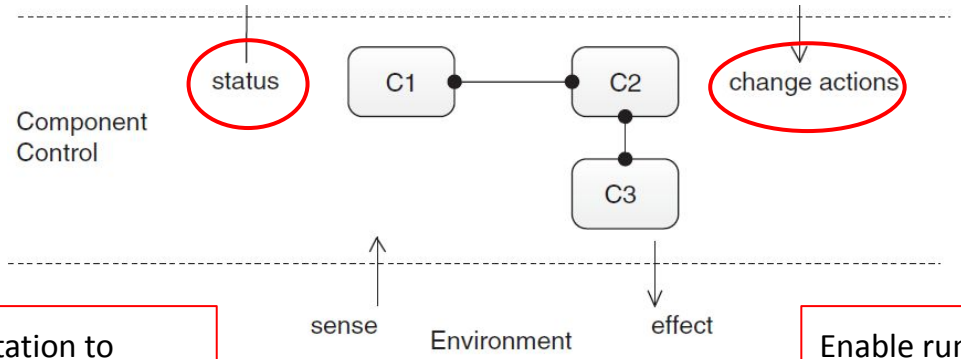
Component Control Layer (1)

Handles runtime events using built-in techniques such as exceptions to maintain normal execution



Components sense and effect the environment in order to realize the goals for the users of the system

Component Control Layer (2)

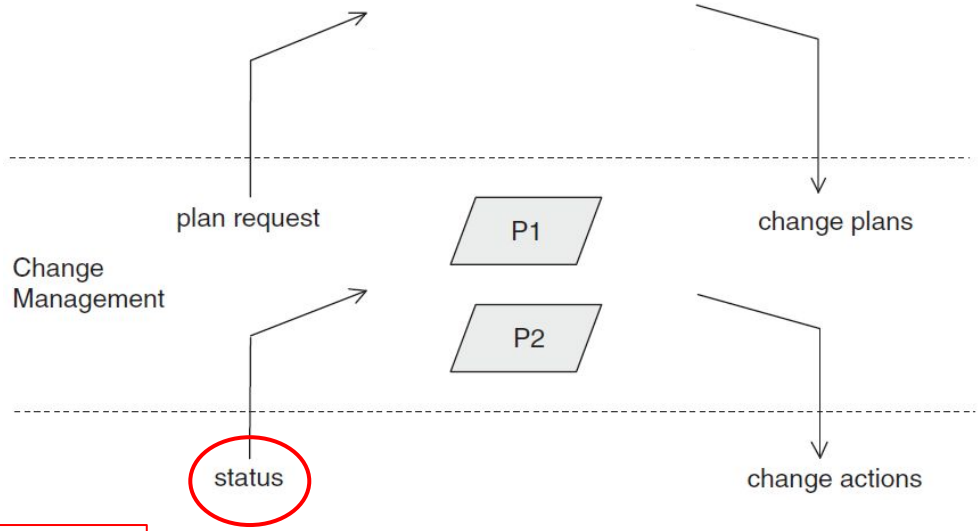


Equipped with instrumentation to report the system's current status to higher layers

Enable runtime modifications (adding, removing, reconnecting, or reconfiguring components).

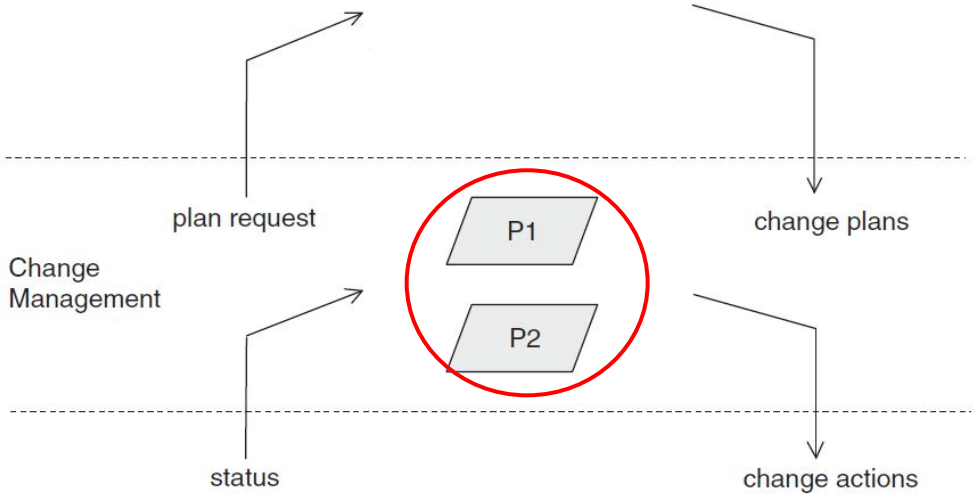
Change Management Layer (1)

Reactive Approach



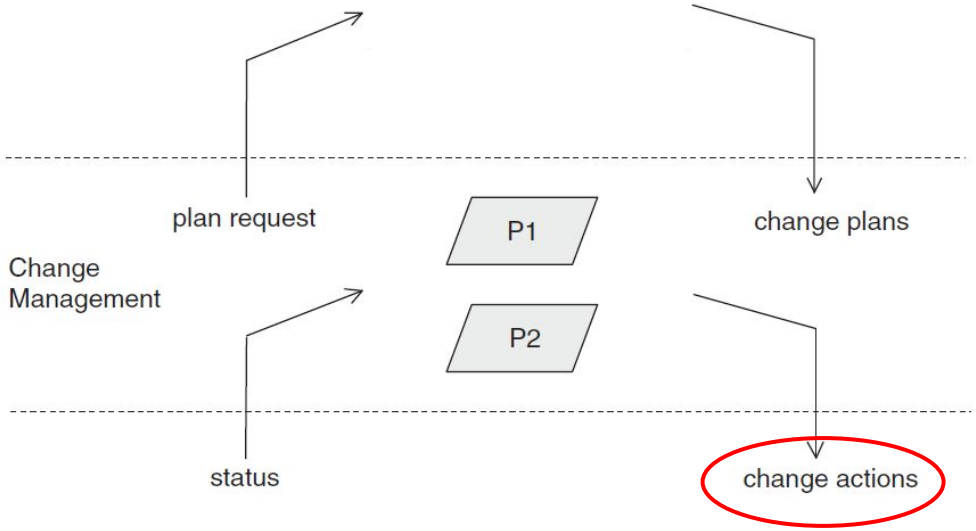
Change management reacts in response to **status changes** of the bottom layer by analyzing the changes

Change Management Layer (2)



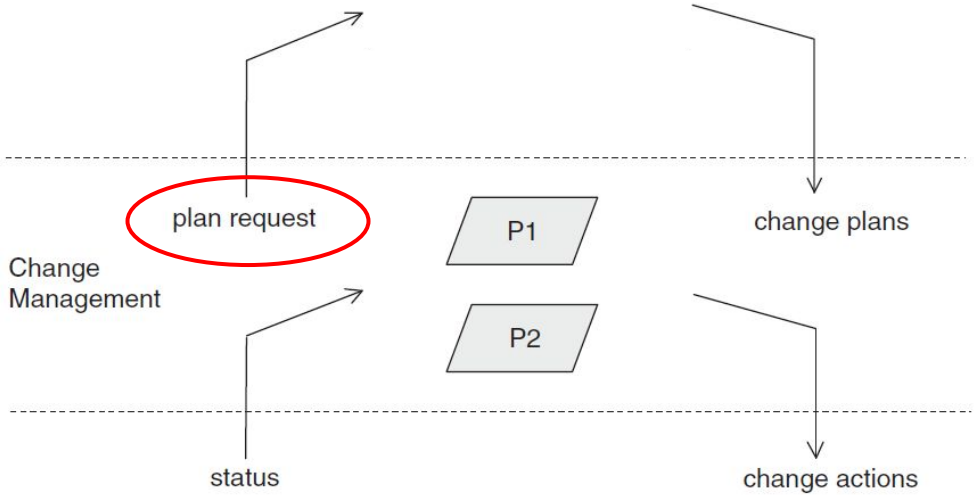
Centered on a set of **plans**, which are typically predefined. Each plan defines a strategy for adapting the system (e.g., adjust parameters, replace components)

Change Management Layer (3)



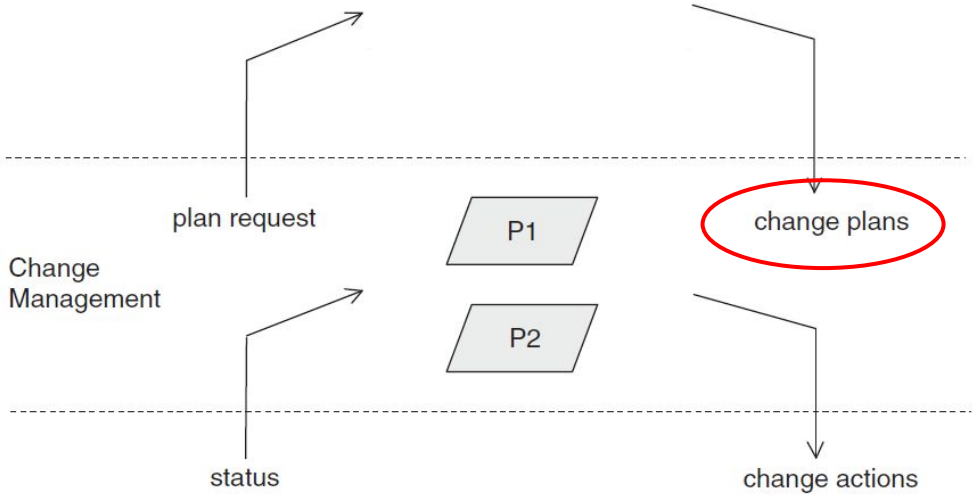
Executes plans through **change actions** that adapt the configuration of the bottom layer. These actions perform the concrete adaptation: tuning parameters, adding/removing components, changing links

Change Management Layer (4)



If a condition is reported that cannot be handled by the available plans, the change management layer invokes the services of the goal management layer

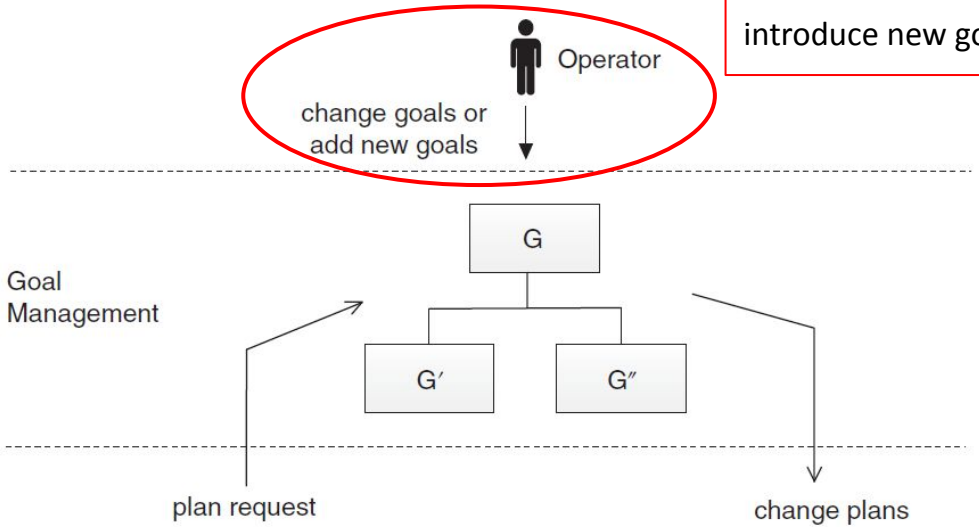
Change Management Layer (5)



The upper layer provides new or updated plans in response to the request

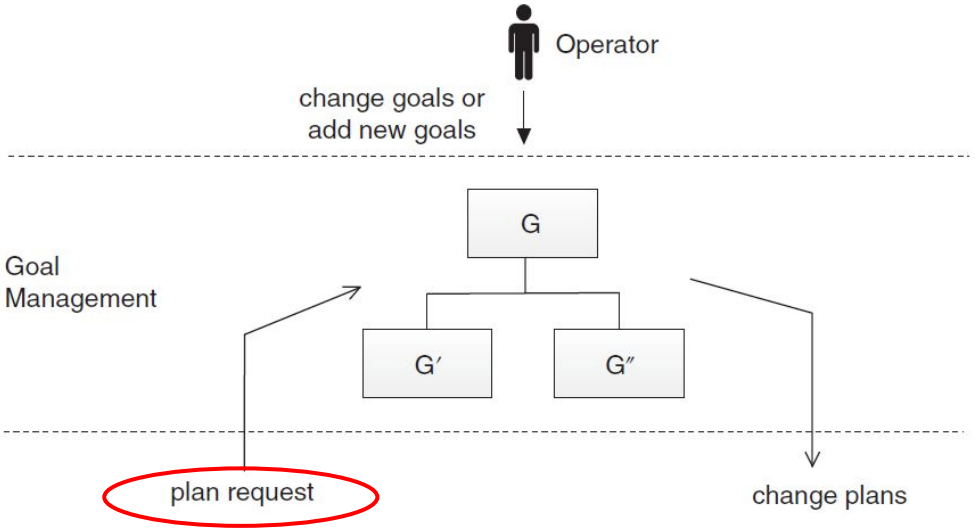
Goal Management (1)

Goal management can also be triggered by stakeholders that want to change goals or introduce new goals



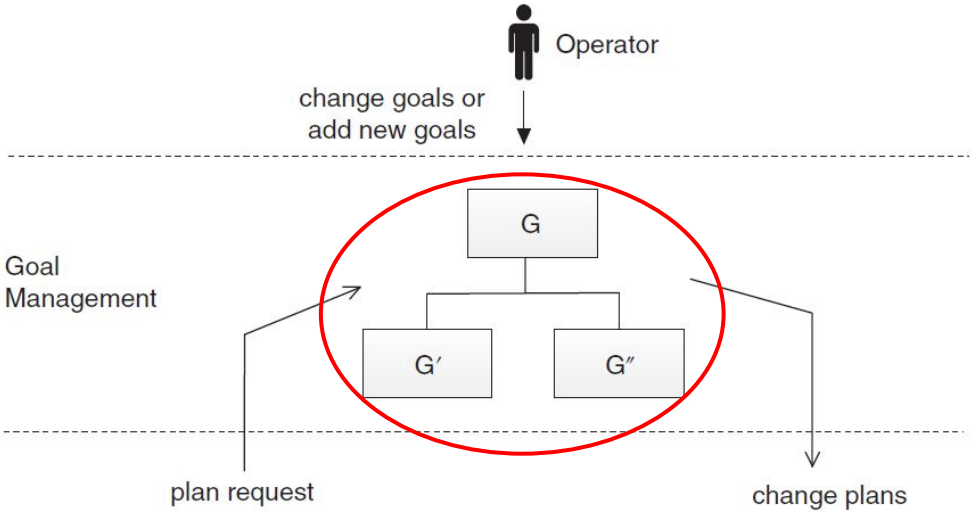
Deliberative Approach

Goal Management (2)



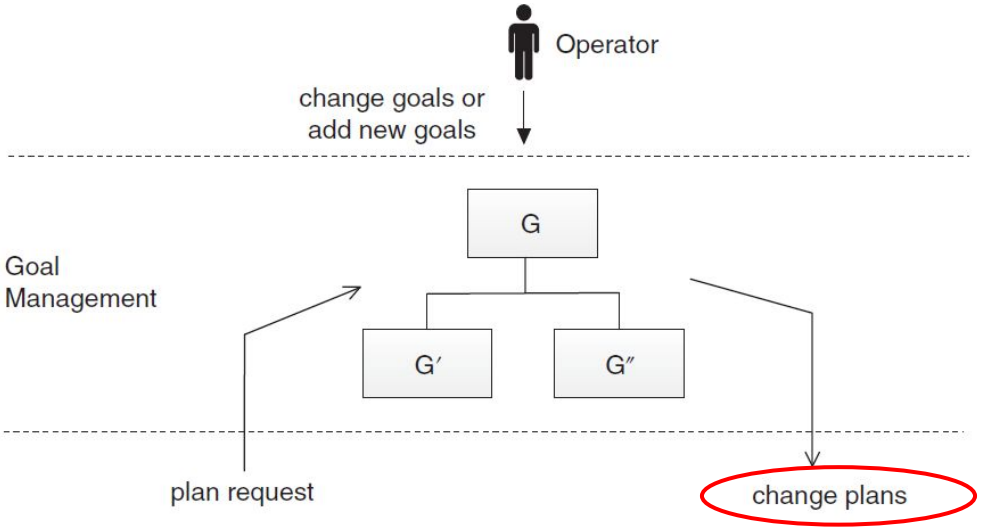
A request from change management will trigger goal management to analyze the situation

Goal Management (3)



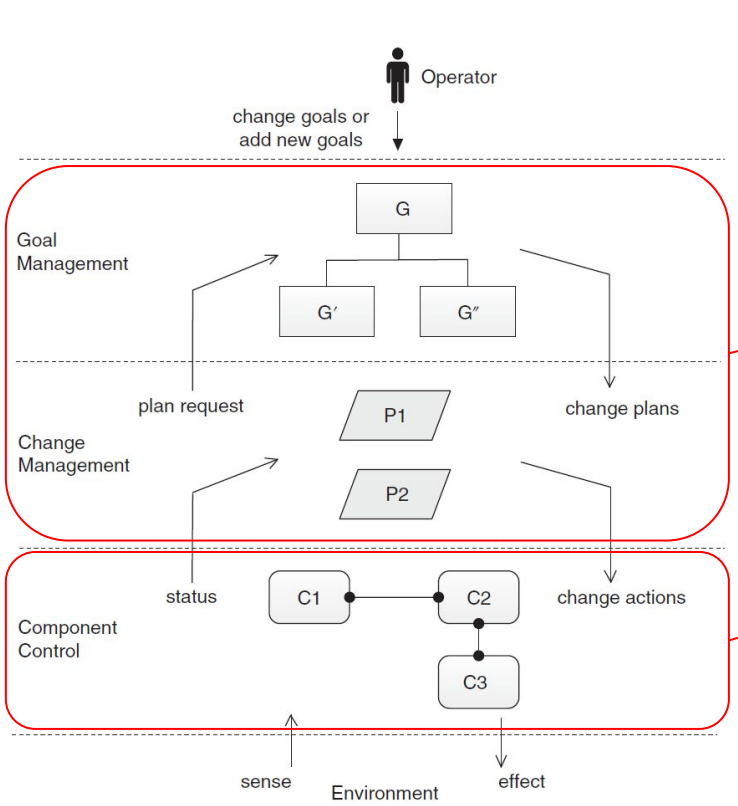
Select alternative goals based on the current status of the system. Instantiated goals or alternative goal sets generated by the reasoning in Goal Management

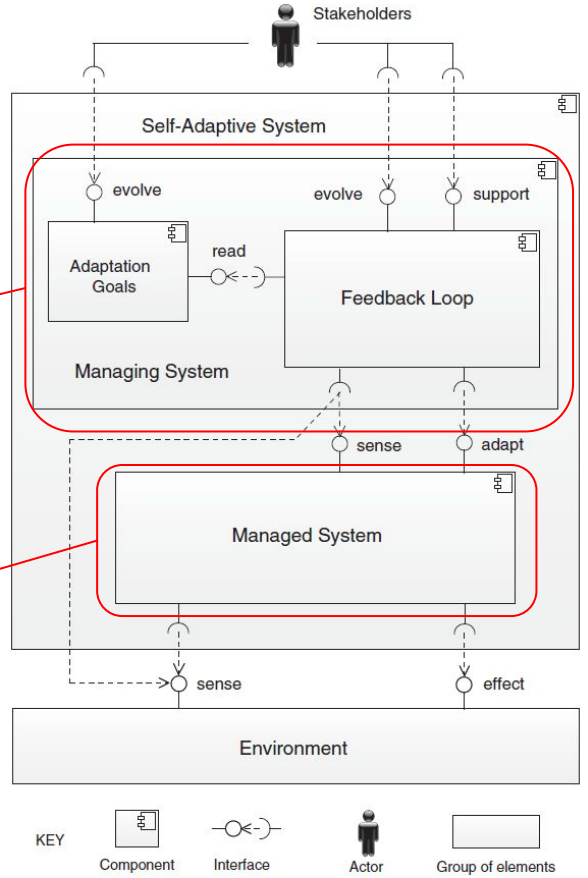
Goal Management (4)



Generate plans to achieve these alternative goals. The new plans are then delegated to the change management layer

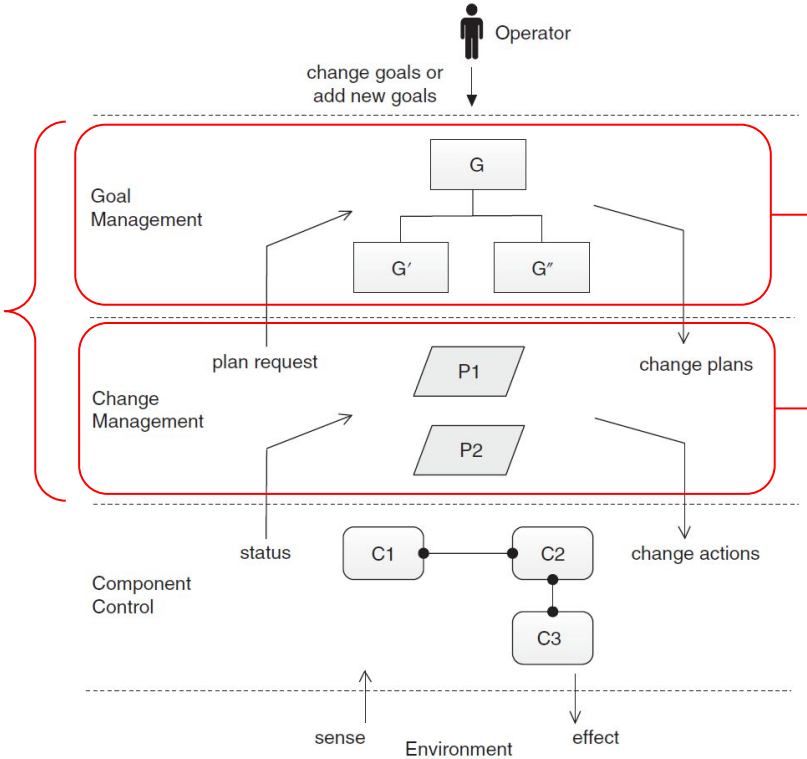
Mapping Between the Three-Layer Model and the Conceptual Model for Self-Adaptation





Mapping Between the Three-Layer Model and the Conceptual Model for Self-Adaptation

Makes an explicit distinction between the functionality of the managing system

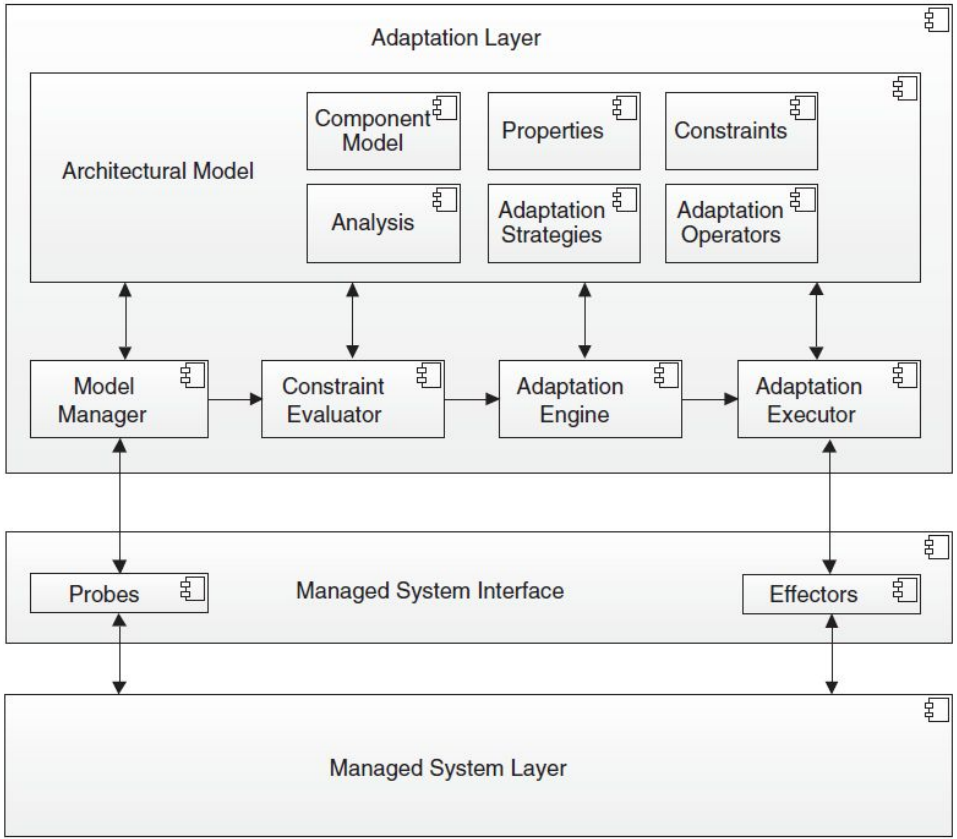


Realizing the adaptation goals given the current conditions

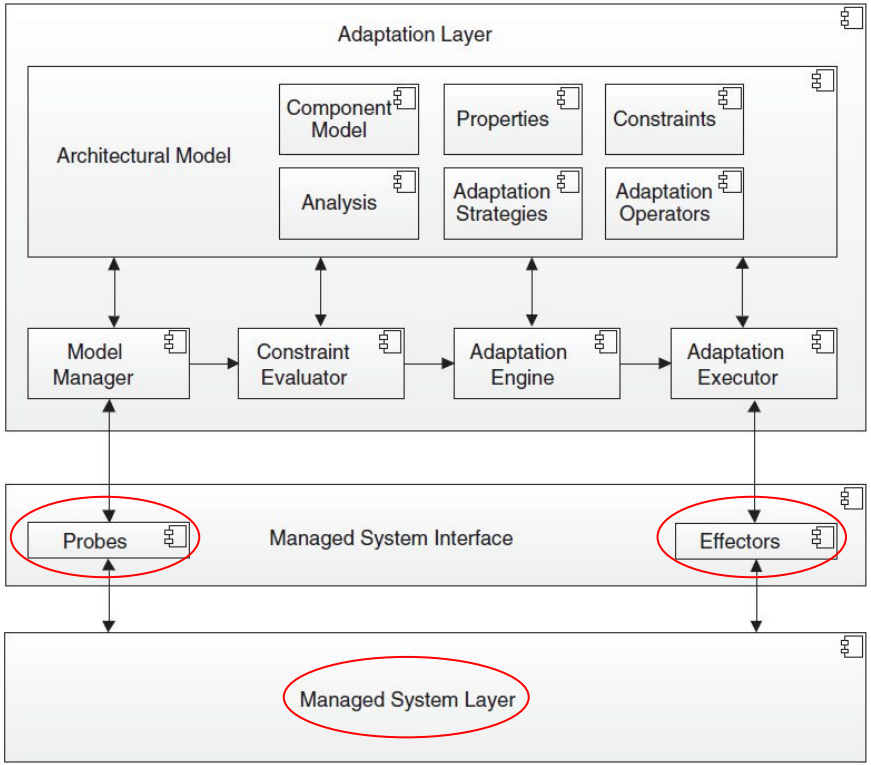
Reasoning about and selecting the adaptation goals

Runtime Architecture of Architecture-based Adaptation (1)

Architectural model to realize self-adaptation



Runtime Architecture of Architecture-based Adaptation (2)



Observing the system

Applying changes to the system

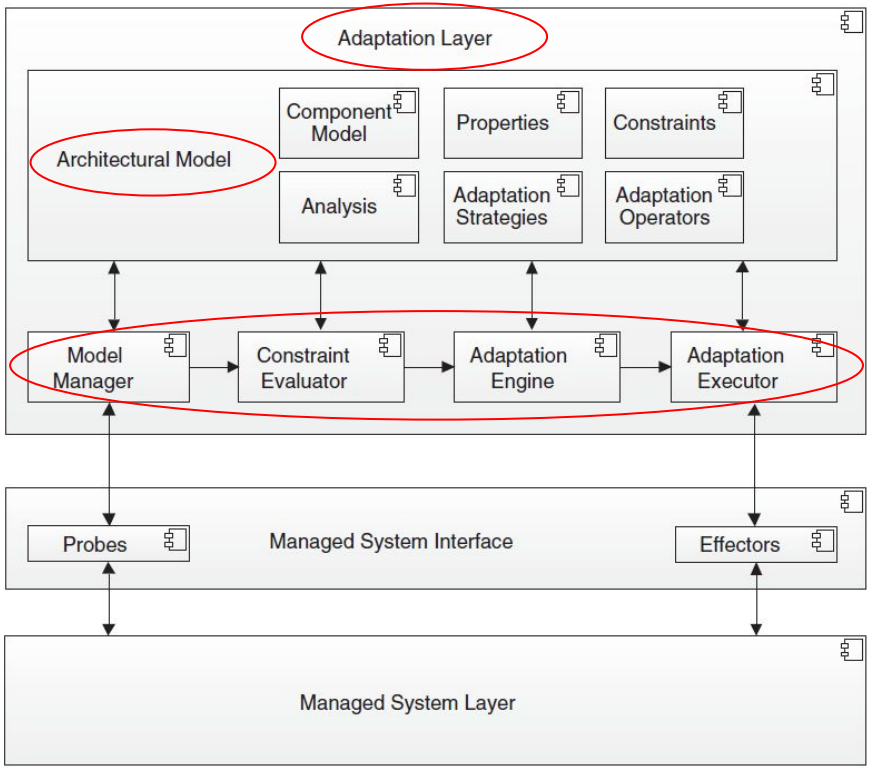
Provides domain functionalities to the users

Runtime Architecture of Architecture-based Adaptation (3)

Components forming a feedback loop

Encodes domain-specific knowledge of the system

Implement the basic functions of a managing system



Architecture-based Adaptation of the Web-based Client-Server System (1)

Web-based client-server system with multiple server groups. Clients send stateless requests to servers via network links. Each server group manages incoming requests through a request queue.

Adaptation Focus:

- **Goal**: Optimize performance, specifically client response time
- **Key influencing factors**: Server load, Network bandwidth, Number of incoming requests (dynamic & unpredictable)

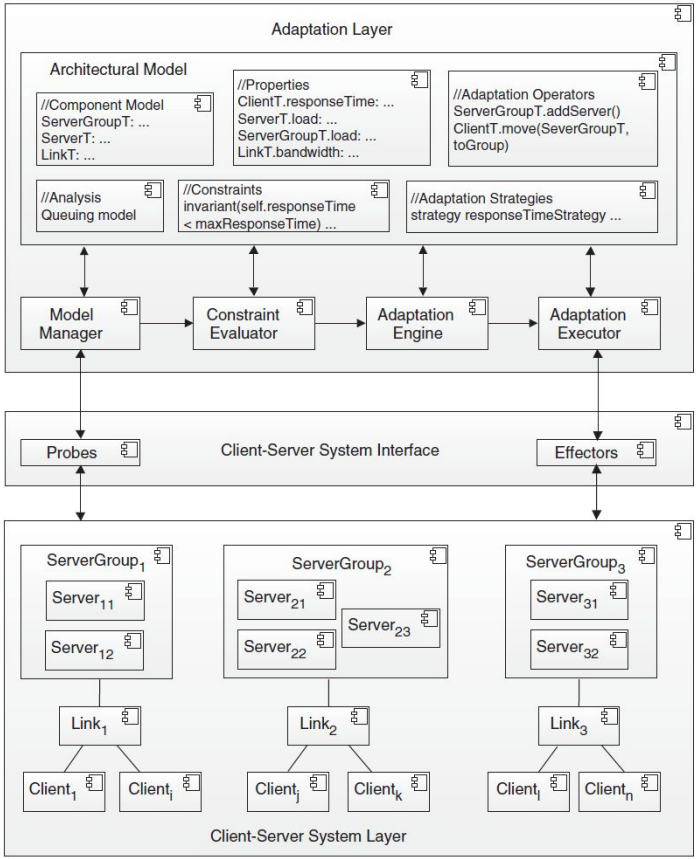
Monitor & Control:

- ClientType.responseTime
- ServerType.load
- LinkType.bandwidth

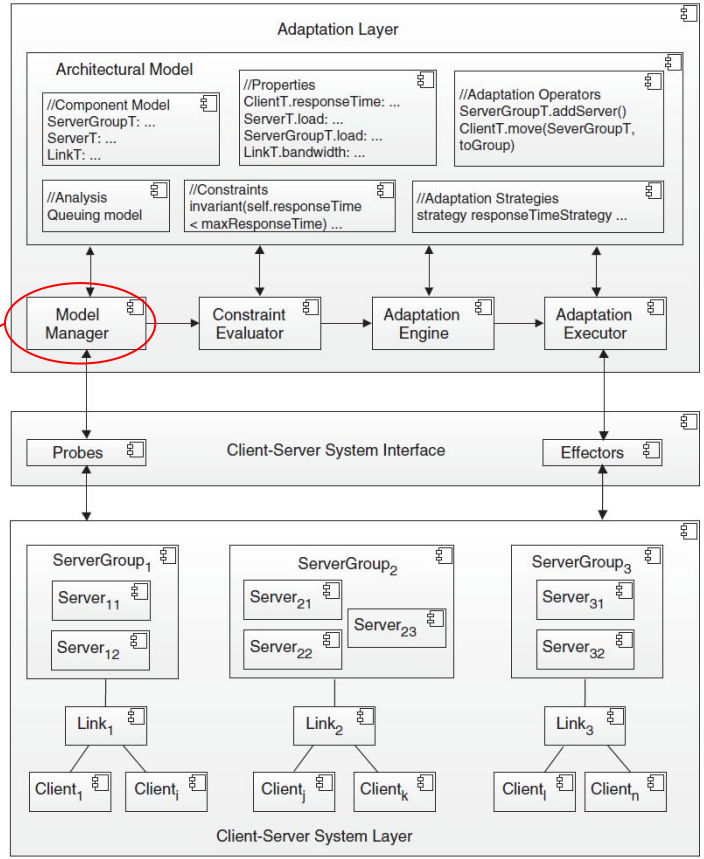
Adaptation goal (constraint for each client):

$$invariant\ (self.responseTime < maxResponseTime)\ !\rightarrow responseTimeStrategy(self);$$

Architecture-based Adaptation of the Web-based Client-Server System (2)

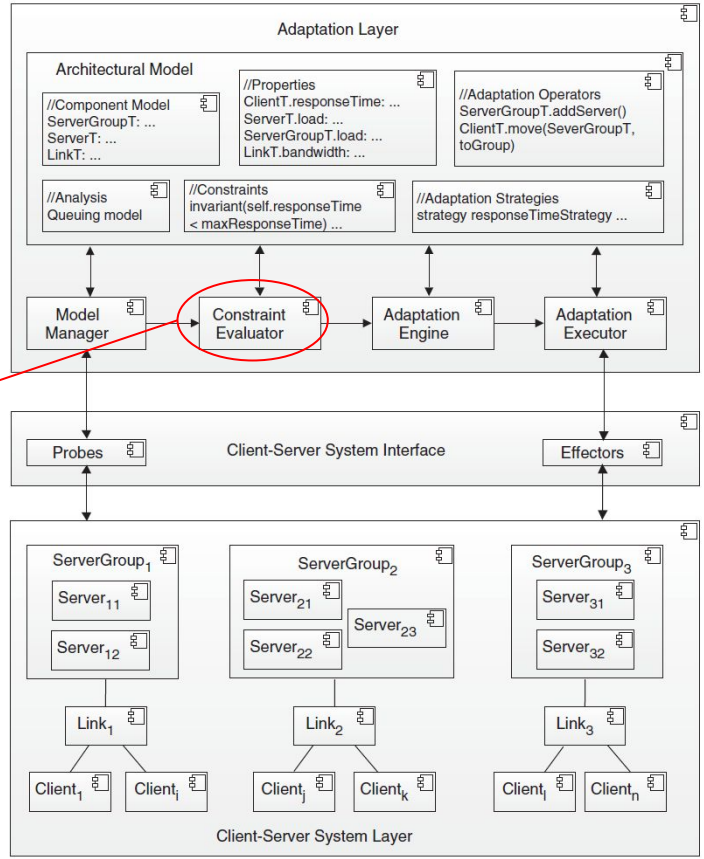


Architecture-based Adaptation of the Web-based Client-Server System (2)



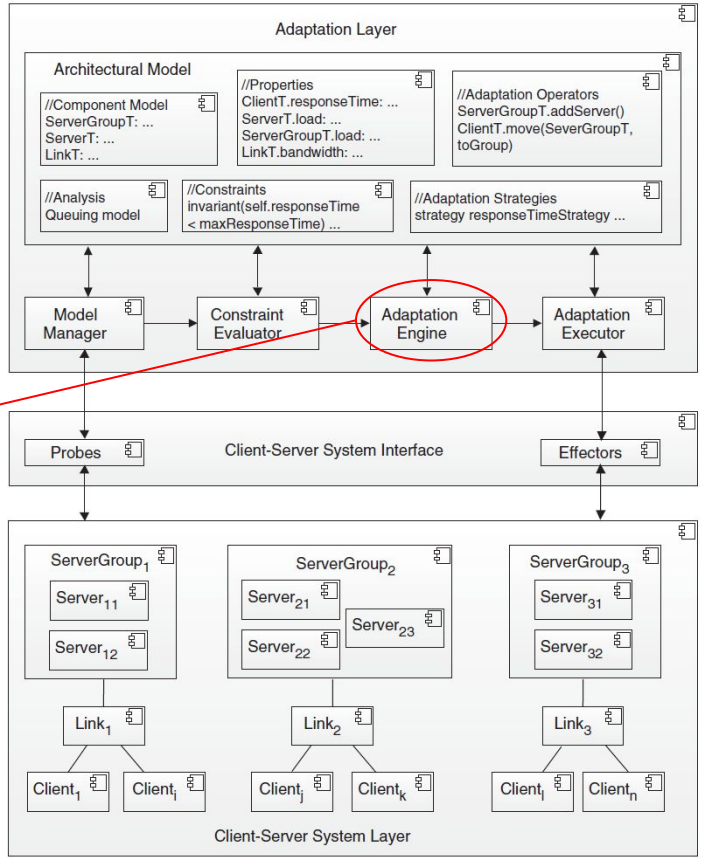
Keeps track of the response time of clients, the server load, and the bandwidth of links

Architecture-based Adaptation of the Web-based Client-Server System (2)



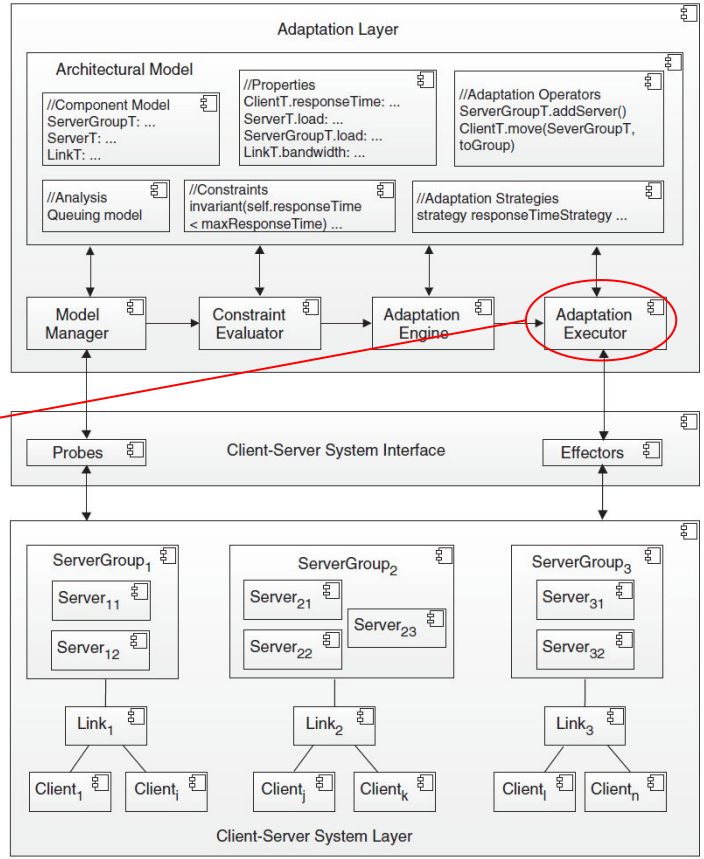
Periodically checks whether the measured response time of clients is below a required threshold bandwidth of links

Architecture-based Adaptation of the Web-based Client-Server System (2)



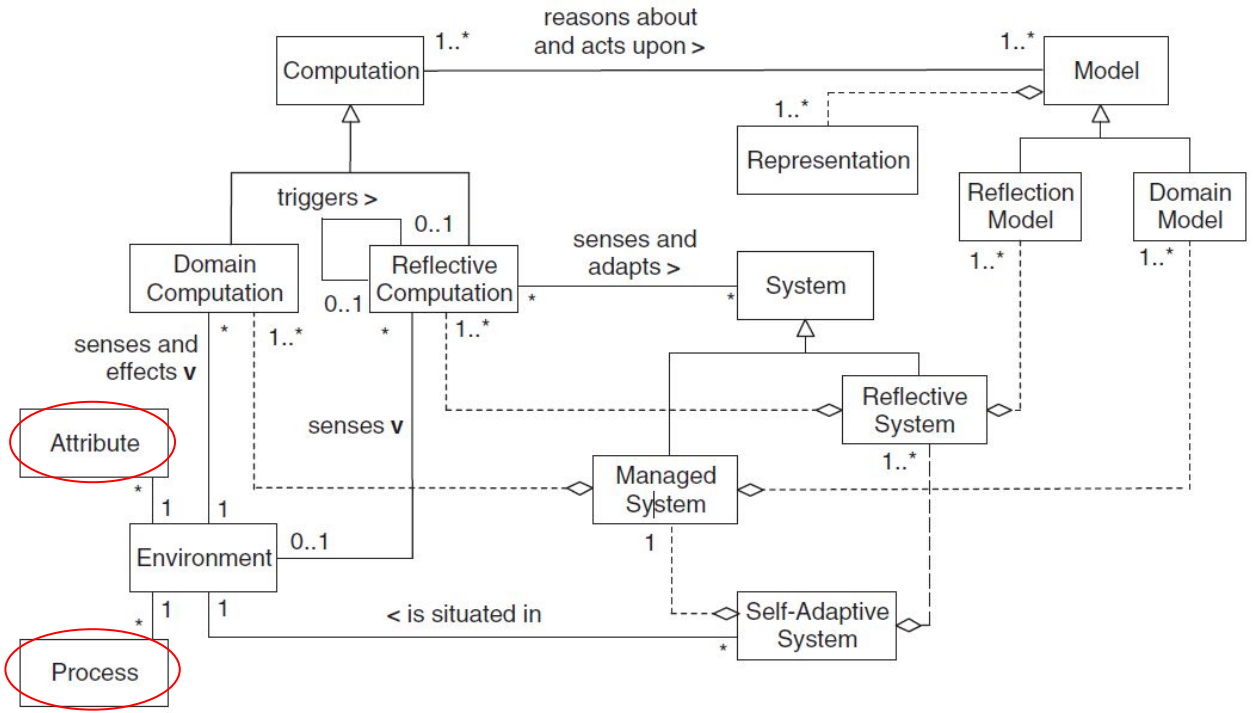
Executes the response time strategy adding a server to the group, decreasing the load and consequently also the response time

Architecture-based Adaptation of the Web-based Client-Server System (2)



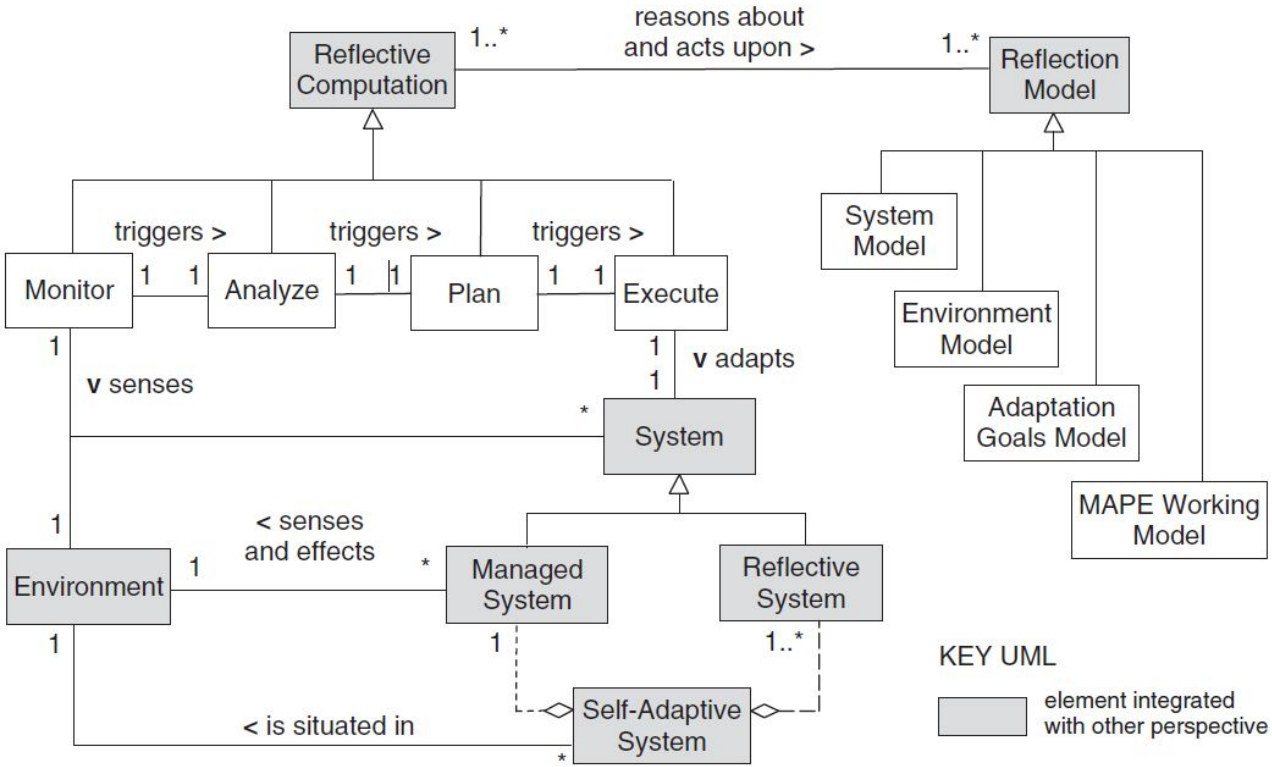
Applies the required operators to adapt the system, i.e. adding a server to the group if the load of the current server group is too high

Comprehensive Reference Model for Self-Adaptation



Perceivable characteristic of the environment

Activity that can change the environment attributes


Reflective Computation
1..*
reasons about and acts upon >
1..*
Reflection Model
triggers >
triggers >
triggers >
Monitor
Analyze
Plan
Execute
System Model
Environment Model
Adaptation Goals Model
MAPE Working Model
v senses
v adapts
System
< senses and effects
Environment
Managed System
Reflective System
< is situated in
Self-Adaptive System
KEY UML
element integrated with other perspective

Wave III

Architecture-based Adaptation

The Third Wave of Self-Adaptation

The third wave centers on runtime models, models used by the system during operation to reason, decide, and adapt

Purpose:

1. Manage the **complexity** of concrete designs
2. Support autonomous **decision-making**
3. Keep system understanding **up to date** during runtime

Runtime Models – From Design to Operation

Offline

Model-driven engineering

Engineers refine models before deployment

Online

Runtime models

systems refine and update models during execution

Benefits:

1. Enable systems to reason about themselves and their context
2. Allow continuous adaptation to changing conditions
3. Bridge the gap between design-time abstraction and runtime reality

What is a Runtime Model?

A runtime model is a first-class runtime abstraction of a system, or any aspect related to it, used to realize self-adaptation

- Managed System Model
- Environment Model
- Adaptation Goals Model
- MAPE Working Model

Used both as **shared knowledge** among MAPE components and as **active elements** in executing the adaptation loop

Causality in Runtime Models

Runtime models must stay consistent with the system or aspects they represent, especially as the system changes dynamically

Outdated models can lead to poor or inaccurate adaptation decisions

Casual connection → ensures that if the system changes, the representation of the system also changes, but also if the representation changes, the system changes accordingly

Real-time synchronization

Full controllability

Weak Causality

Issues of “strong” causality

1. Delay in the update of model or system
2. Runtime models may not represent the ground truth when dealing with uncertainty

Weak causality Links the state of a runtime model and the system, allowing an acceptable discrepancy (temporal, quantitative, etc.)

Motivations for Runtime Models (1)

Enable systems to adapt intelligently at runtime through model-based reasoning and abstraction

Representing Dynamic Change:

1. Capture system evolution at an abstract, system-wide level
2. Focus on relevant runtime aspects, omitting low-level details

Separation of Concerns:

1. Different models for different system aspects
2. Enable tracking, understanding, prediction, and planning

Runtime Reasoning:

1. Support monitoring, constraint checking, simulation, and “what-if” analysis
2. Enable formal reasoning for reliable adaptation decisions

Motivations for Runtime Models (2)

Leveraging Humans in the Loop:

1. Models designed for human readability (e.g., domain-specific languages)
2. Support human participation in:
   a. Data enrichment
   b. Evaluation and decision-making
   c. Guiding large-scale adaptation

Facilitating On-the-Fly Evolution:

1. Runtime models act as living design artifacts
2. Enable detection of issues and live updates during operation
3. Models can evolve and be redeployed dynamically

Dimensions of Runtime Models

Runtime models can be classified according to different **dimensions**

Dimension → describes a particular characteristic of the representation of a self-adaptive system or aspects related to the system

Options → represent the extremes of the domain

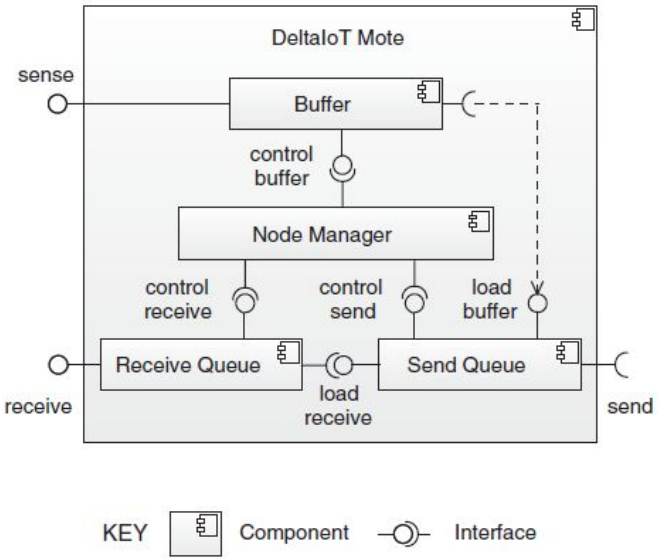
Structural vs. Behavioral Runtime Models

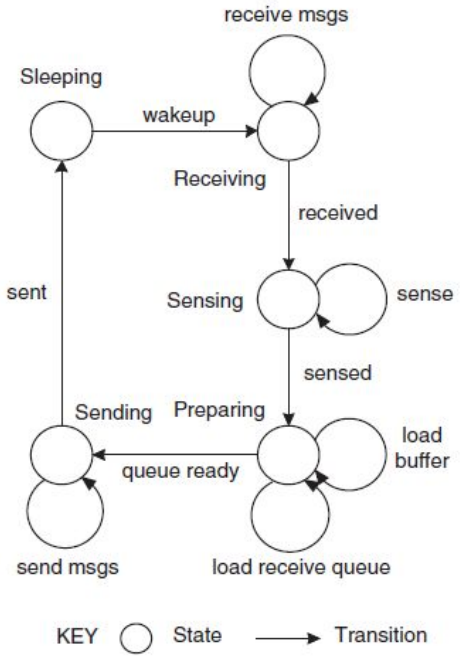
Structural Models

- Focus on **system composition** (elements and their relationships)
- Represent **types or instances** at varying abstraction levels
- Capture **what** the system is (its architecture and configuration
- Useful for reasoning about **system organization and structure changes**

Behavioral Models

- Focus on **dynamic behavior** and **changes over time**
- Describe **Activities** (event sequences, control/data flows)
- Describe **State changes** (transitions, protocols, interactions)
- Capture **how** the system behaves and evolves
- Useful for analyzing **runtime processes** and **interaction dynamics**


DeltaIoT Mote
sense
Buffer
control buffer
Node Manager
control receive
control send
load buffer
Receive Queue
Send Queue
receive
load receive
send
KEY
Component
Interface


receive msgs
Sleeping
wakeup
Receiving
received
sent
Sensing
sense
sensed
Sending
Preparing
load buffer
queue ready
send msgs
load receive queue
KEY
State
Transition

Declarative vs. Procedural Runtime Models

Declarative Models

- Describe **what** needs to be achieved or **what is true**
- Capture **goals, constraints, or desired states**
- Focus on **purpose and outcomes**, not steps

Example: an **adaptation goal** defining system performance targets

Procedural Models

- Describe **how** something is or should be done
- Capture **processes, plans, and actions** for adaptation
- Focus on **execution and method**

Example: a **plan** specifying steps to adapt the system

Declarative model

Procedural model

```
//Declaration of the adaptation goals
int MAX_PACKET_LOSS = 10; //max packet loss 10%
...
//Procedures to test the adaptation goals for a given configuration
bool satisfactionGoalPacketLoss(Configuration gConf,
  int MAX_PACKET_LOSS) {
    return gConf.qualities.packetLoss < MAX_PACKET_LOSS;
}
...
```

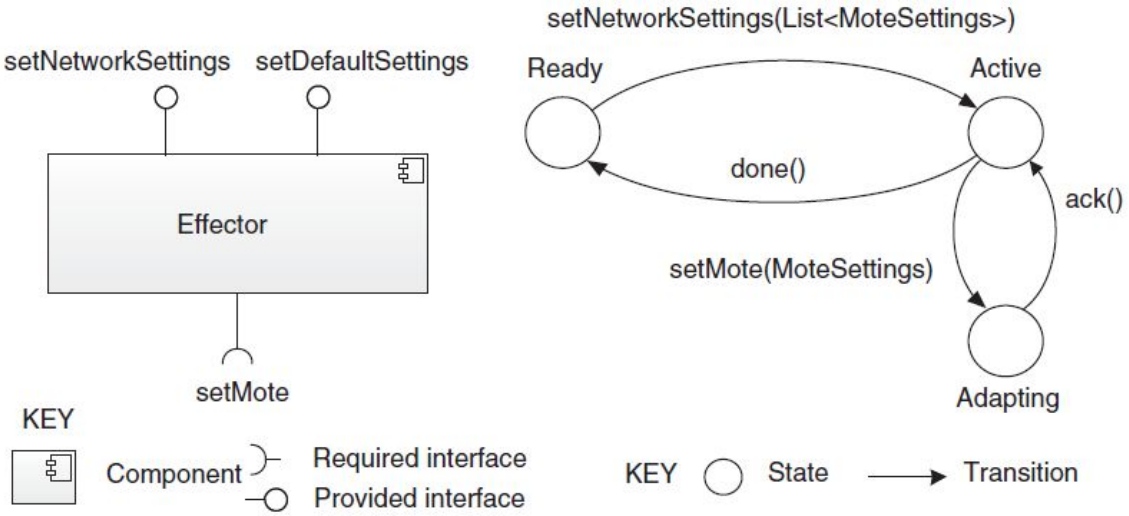
Functional vs. Qualitative Runtime Models

Functional Models

- Represent the **functionality** of the system or its components
- Describe **elements, their functions, and interrelations**
- Capture **inputs, outputs, and data flows**
- Support **functional analysis** e.g., checking validity of system reconfigurations
- Ensure the system continues to provide **required services after adaptation**

Verify what the system does and how its functions interact

Qualitative Models


setNetworkSettings
setDefaultSettings
Effector
setMote
KEY
Component
Required interface
Provided interface
setNetworkSettings(List<MoteSettings>)
Ready
Active
done()
ack()
setMote(MoteSettings)
Adapting
KEY
State
Transition

Qualitative Runtime Models

Defined by ISO 9126/25010:

“the set of characteristics, and the relationships between them, that provides the basis for specifying quality requirements and evaluation.”

Key Properties:

- **Discretization**: represent continuous aspects via discrete values for reasoning
- **Relevance**: focus on properties meaningful for decision-making
- **Accuracy**: maintain precision despite uncertainty

Examples:

- **Markov Models**: stochastic state–transition models for reliability prediction
- **Queuing Models**: analyze performance based on workload and response time

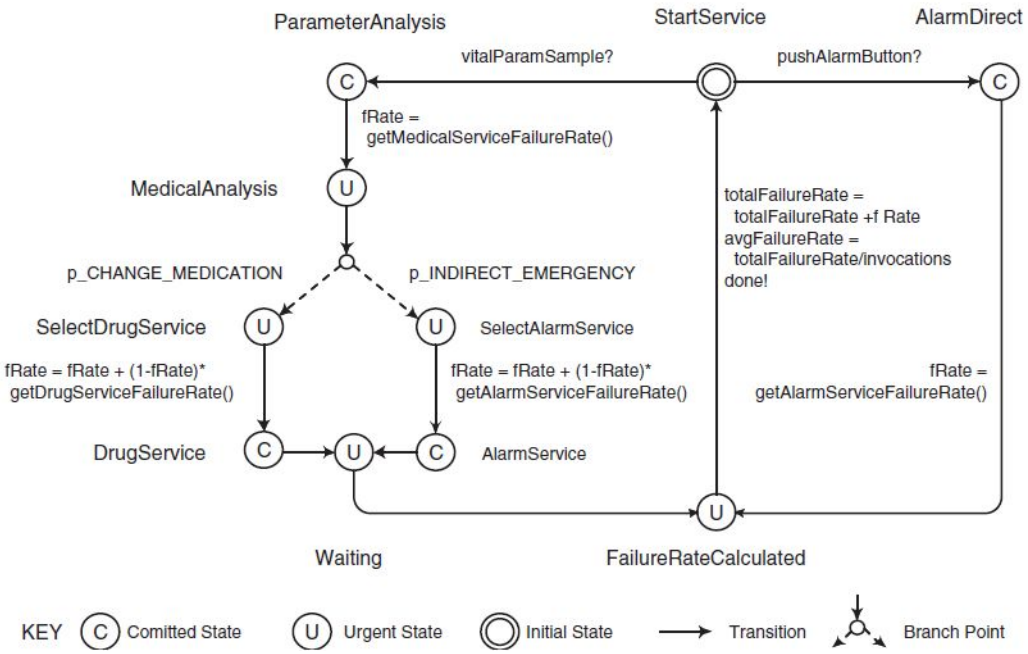
Formal vs. Informal Runtime Models

Formal Models

- Based on **mathematical foundations** (discrete math, logic, automata)
- Have **well-defined syntax and semantics**
- Enable **automated reasoning** and **guarantees** for strict adaptation goals
- Produce **replicable** analytical results
- **Limitations:** hard to capture all real-world aspects; modeling can be complex

Informal Models

- Use **domain abstractions** or **programming-oriented notations** without full formal basis
- Easier and faster to apply; suitable when formalization is **impractical or too costly**
- May include **stakeholder input** or **heuristic reasoning**
- Less precise → may involve **interpretation or ambiguity**
- Practical for **real-world adaptation** when full rigor isn't required


ParameterAnalysis
StartService
AlarmDirect
vitalParamSample?
pushAlarmButton?
C
fRate =
getMedicalServiceFailureRate()
MedicalAnalysis
U
totalFailureRate =
totalFailureRate +f Rate
avgFailureRate =
totalFailureRate/invocations
done!
p_CHANGE_MEDICATION
p_INDIRECT_EMERGENCY
SelectDrugService
SelectAlarmService
fRate = fRate + (1-fRate)*
getDrugServiceFailureRate()
fRate = fRate + (1-fRate)*
getAlarmServiceFailureRate()
fRate =
getAlarmServiceFailureRate()
DrugService
AlarmService
Waiting
FailureRateCalculated
KEY
Comitted State
Urgent State
Initial State
Transition
Branch Point

Principal Strategies for Using Runtime Models

How a runtime model can be used? 3 strategies

Shared Model Strategy

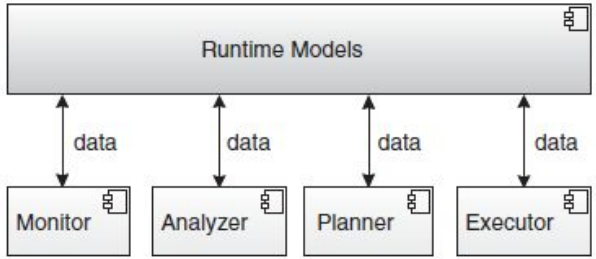
All MAPE components access a **common set of runtime models**. Most frequently used in self-adaptive systems. Promotes **consistency** and **centralized knowledge**

Exchanged Model Strategy

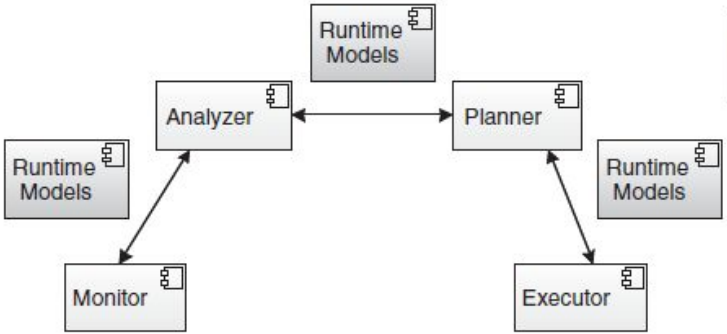
MAPE components **exchange only the models they need.** Reduces **data sharing overhead.** Less commonly used

Shared MAPE Model Strategy

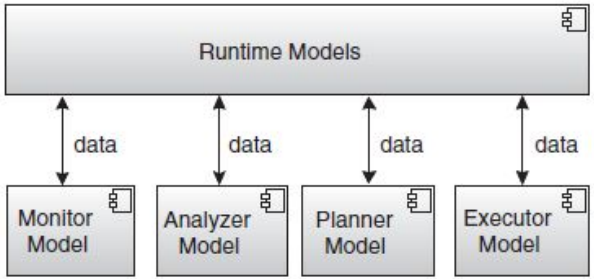
MAPE models themselves **share a common runtime model set.** Supports modular design of MAPE components. Less frequently applied



(a) MAPE Components Share K Models



(b) MAPE Components Exchange K Models



(c) MAPE Models Share K Models

MAPE Components Share Models

All MAPE components access a **common knowledge repository.** Centralizes **runtime models** for the managed system and environment

Workflow:

1. **Monitor** → updates **runtime models** with probe data (system + environment)
2. **Analyzer** → queries **models** via runtime simulation; evaluates adaptation options. Configures **parameterized quality models**
3. **Planner** → reads analysis results from **models** to select adaptation option. Generates **adaptation plan** and writes it to **models**
4. **Executor** → reads **plan from models** and applies it to the system

MAPE Components Exchange Models

MAPE components **exchange runtime models** instead of using a single shared repository. Components reason on and manipulate models relevant to their function. Adaptation emerges as a result of model exchanges and updates

System Architecture (Three Layers):

1. **Business Application (Bottom Layer)**
   - Managed system with **application components**
   - Equipped with **sensors** and **factories** for runtime monitoring and instantiation

2. **Causal Connection (Middle Layer)**
   - Implements **weak causal connection** between runtime models and managed system

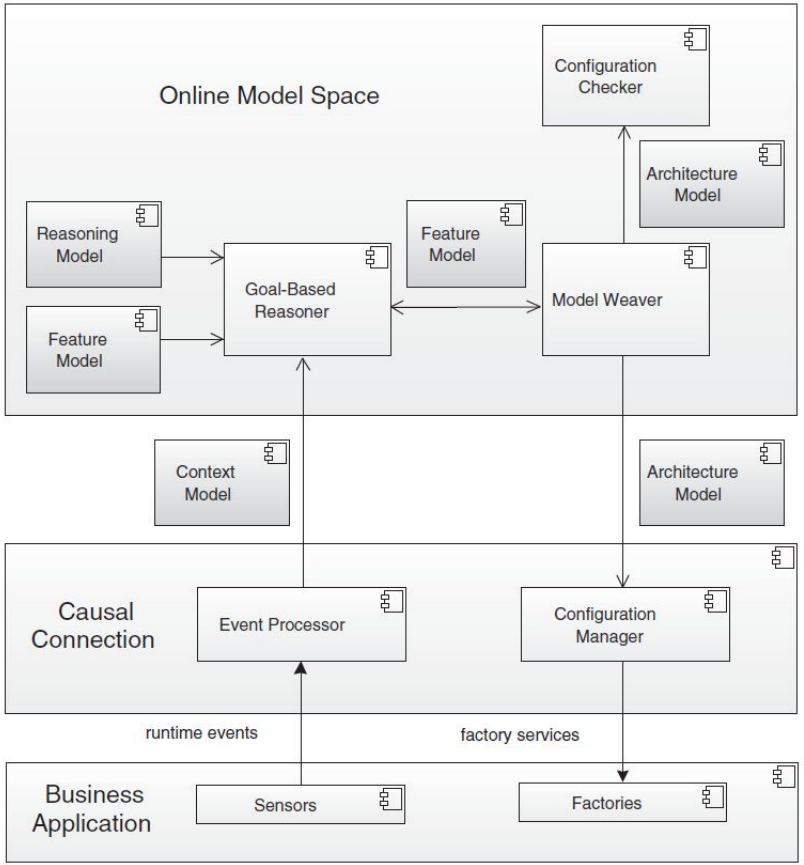
3. **Online Model Space (Top Layer)**
   - MAPE components manipulate **runtime models**
   - Five adaptation components exchange **four types of runtime models**

System Architecture (Three Layers)

1. **Business Application (Bottom Layer)**
   - Managed system with **application components**
   - Equipped with **sensors** and **factories** for runtime monitoring and instantiation

2. **Causal Connection (Middle Layer)**
   - Implements **weak causal connection** between runtime models and managed system

3. **Online Model Space (Top Layer)**
   - MAPE components manipulate **runtime models**
   - Five adaptation components exchange **four types of runtime models**



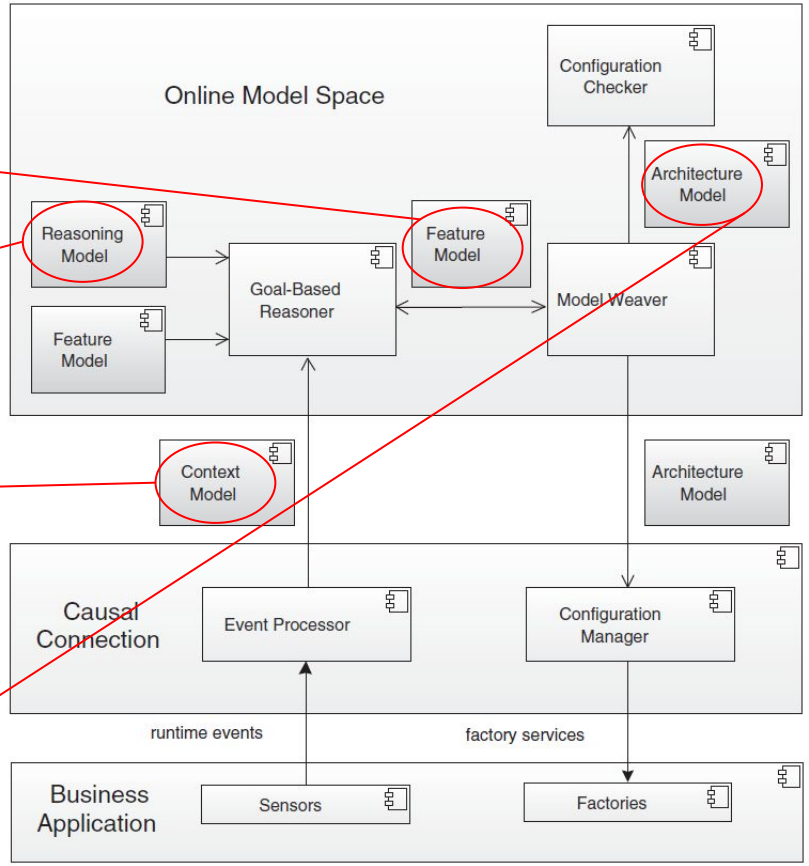
System Architecture Runtime Models

Specifies the system's variability, including mandatory, optional, and alternative features, along with constraints among them, linking features to architectural fragments

Associates sets of features with contexts, often as event-condition-action rules, to trigger adaptation actions based on system events and condition

Captures relevant environmental attributes and processes, kept up-to-date at runtime using sensor data to inform adaptations

Defines the component composition of the business application, supporting dynamic reconfigurations and refining feature model leaves into concrete architectural fragments



MAPE Models Share Models

Runtime models are **shared by MAPE functions themselves**. MAPE functions are **specified and executed as runtime models**, making the adaptation logic fully model-centric

MAPE Models Share Models

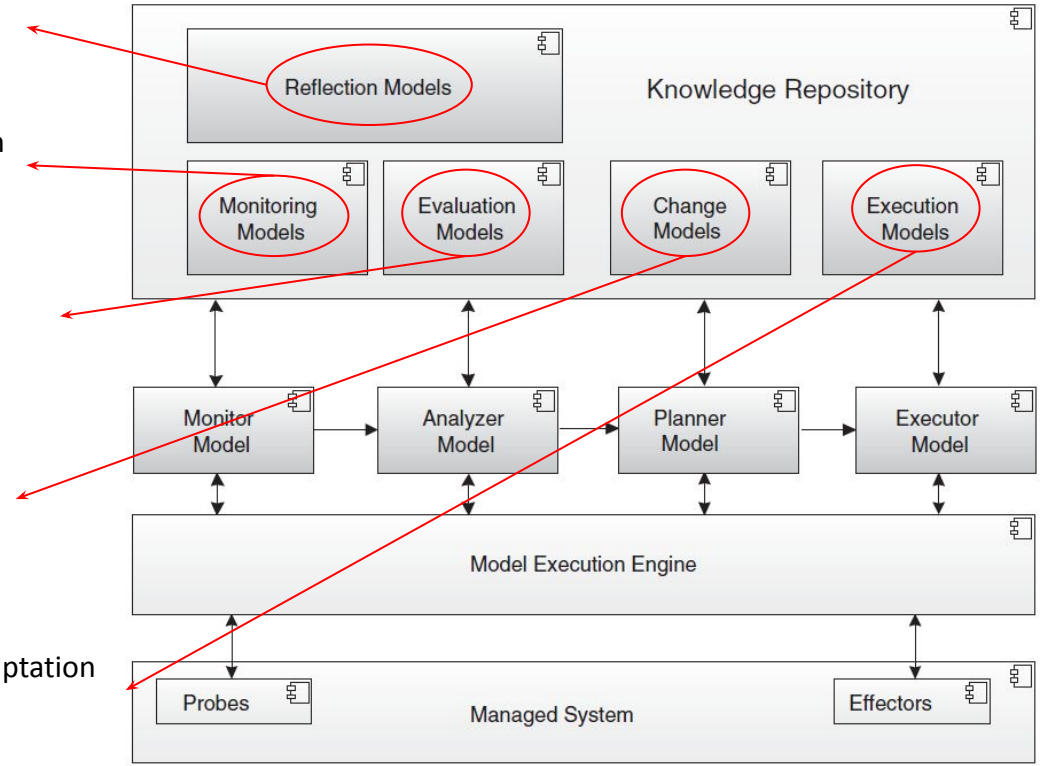
Represent architectural aspects of the managed system and environment

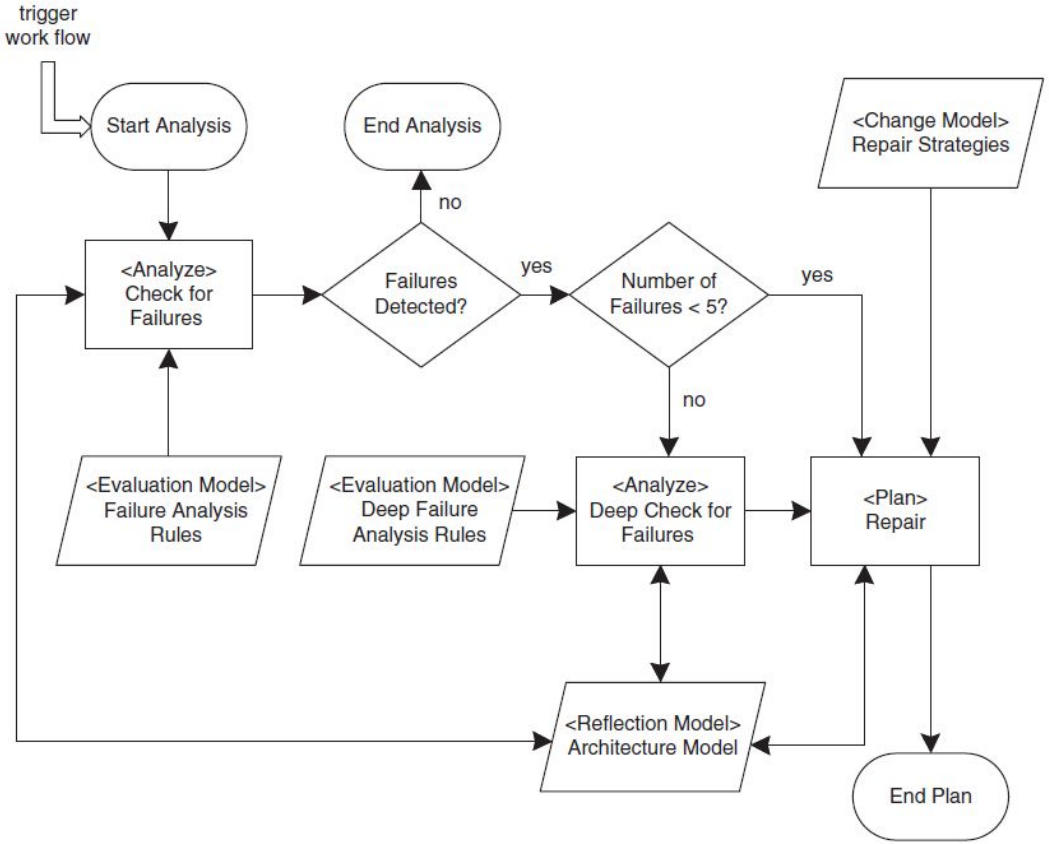
Map system observations to the abstraction used in reflection models

Define allowed configurations and thresholds on quality properties

Define the space of allowed system variability for planning

Map plan-level steps to concrete system-level adaptation actions




trigger
work flow
Start Analysis
End Analysis
<Change Model>
Repair Strategies
no
<Analyze>
Check for
Failures
Failures
Detected?
yes
Number of
Failures < 5?
yes
no
<Evaluation Model>
Failure Analysis
Rules
<Evaluation Model>
Deep Failure
Analysis Rules
<Analyze>
Deep Check for
Failures
<Plan>
Repair
<Reflection Model>
Architecture Model
End Plan

Wave IV

Requirements-driven Adaptation

Requirements-driven Adaptation

Focuses on requirements understanding stakeholder needs and linking them to adaptive behavior. Self-adaptive systems must be built on a clear understanding of requirements, not ad-hoc feedback loops

Adaptation goals express stakeholder concerns in a machine-readable, operational form to guide system adaptation

Requirements Engineering Focus

Key activities:

1. **Elicitation** ⟶ Identify stakeholders' needs, system goals, and adaptation conditions
2. **Analysis** ⟶ Examine and prioritize requirements; resolve conflicts and trade-offs
3. **Specification** ⟶ Clearly define requirements (often as adaptation goals or rules)
4. **Operationalization** ⟶ Link requirements to system elements and feedback loops for execution
5. **Maintenance** ⟶ Update and refine requirements as the system and context evolve

Three Main Approaches

1) Relaxed Requirements → Tolerate uncertainty in dynamic environments

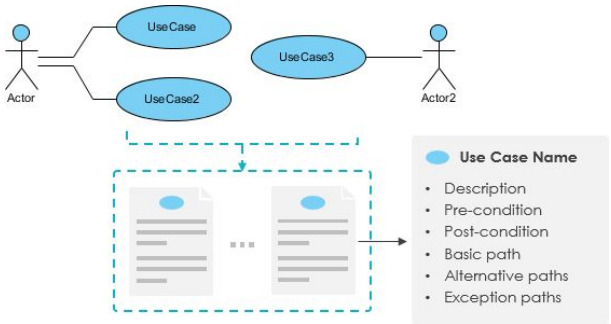
2) Meta-Requirements→ Specify requirements about other requirements to guide adaptive behavior

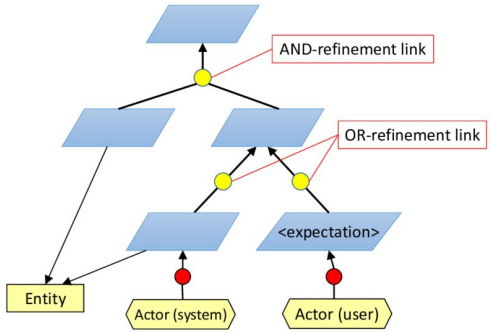
3) Feedback Loop Requirements → Define functional behavior and adaptability of feedback mechanisms

Relaxing Requirements for Self-Adaptation

Traditional Requirements

- Define **what** a system should do (functions) and **how** it should perform (qualities)
- May include **why** the requirements exist (stakeholder rationale)
- Common notations:
  - *Use cases*, *user stories* ⟶ functional requirements
  - *Structured natural language*, *quality scenarios*, *goal models* ⟶ quality requirements





Relaxing Requirements for Self-Adaptation

Challenge for Self-Adaptive Systems

- These systems face **uncertainty** in environment and operation
- Traditional requirement methods are often **too rigid**

Relaxed Requirements Approach

- Uses **structured natural language** to make requirements more **flexible**
- Allows adaptation goals to be **“relaxed”** to handle changing or uncertain conditions.
- Example: Instead of *“Response time must always be < 2s”*, use *“Response time should be < 2s under normal conditions.”*

Relaxed Specification Approach

Introduces a **structured language** to mark:

- **Invariants** → requirements that *must always hold*
- **Relaxable requirements** → can *temporarily change* under certain conditions

Based on the RELAX Language

- Designed specifically for **self-adaptive systems**
- Provides **explicit support** for expressing **uncertainty and flexibility** in requirements

The RELAX Language

Operators	Explanation
Model operators	
SHALL	The requirement must hold
MAY…OR	Requirement with alternative options
Temporal operators	
AS SOON AS POSSIBLE TO	The requirement should hold as soon as possible
IN [interval]	The requirement should hold in a time interval
Ordinal operators	
AS CLOSE AS POSSIBLE TO [quantity]	Relaxation of value of countable quantity specified in a requirement
AS MANY AS POSSIBLE [quantity]	Relaxation of number of countable quantity specified in a requirement

The RELAX Language

Modal Operators: define obligation level (e.g., *SHALL*, *MAY*)

Operators	Explanation
Model operators	
SHALL	The requirement must hold
MAY…OR	Requirement with alternative options
Temporal operators	
AS SOON AS POSSIBLE TO	The requirement should hold as soon as possible
IN [interval]	The requirement should hold in a time interval
Ordinal operators	
AS CLOSE AS POSSIBLE TO [quantity]	Relaxation of value of countable quantity specified in a requirement
AS MANY AS POSSIBLE [quantity]	Relaxation of number of countable quantity specified in a requirement

The RELAX Language

Temporal Operators: add flexibility to when a requirement must hold

Relaxation:

- ***Strict***: *"The system SHALL keep packet loss under a threshold at all times."*
- ***Relaxed***: *"The system SHALL keep average packet loss under a threshold IN 12 hours."*

Operators	Explanation
Model operators	
SHALL	The requirement must hold
MAY…OR	Requirement with alternative options
Temporal operators	
AS SOON AS POSSIBLE TO	The requirement should hold as soon as possible
IN [interval]	The requirement should hold in a time interval
Ordinal operators	
AS CLOSE AS POSSIBLE TO [quantity]	Relaxation of value of countable quantity specified in a requirement
AS MANY AS POSSIBLE [quantity]	Relaxation of number of countable quantity specified in a requirement

The RELAX Language

Ordinal Operators: add flexibility to *quantities or degrees*

Operators	Explanation
Model operators	
SHALL	The requirement must hold
MAY…OR	Requirement with alternative options
Temporal operators	
AS SOON AS POSSIBLE TO	The requirement should hold as soon as possible
IN [interval]	The requirement should hold in a time interval
Ordinal operators	
AS CLOSE AS POSSIBLE TO [quantity]	Relaxation of value of countable quantity specified in a requirement
AS MANY AS POSSIBLE [quantity]	Relaxation of number of countable quantity specified in a requirement

Semantics of Language Primitives

Every specification language **needs precise semantics** to ensure consistent interpretation. For relaxed requirements, semantics are often defined using **fuzzy temporal logic**

Fuzzy Temporal Logic Basics

Temporal Logic: describes properties over system paths and states

Example: **AG p** $\rightarrow$ for *all paths (A)*, *p always (G)* holds

Fuzzy Sets: extend classical logic (true = 1, false = 0) with degrees of truth $\in [0, 1]$

Enables reasoning under uncertainty

Operationalization of Relaxed Requirements

Transform relaxed requirements into self-adaptive system behavior.

Core aspects

- Handling uncertainties
- Requirements reflection
- Mitigation mechanisms

Handling Uncertainty

1. **Identify sources of uncertainty**
   - Determine environmental or behavioral factors that may prevent strict satisfaction of requirements.

2. **Analyze each requirement**
   - Decide if it must *always* hold (**invariant**) or can be *relaxed* under certain conditions.

3. **Document uncertainty conditions**
   - Define when and why relaxation is allowed (e.g., overload, interference, data loss).

4. **Monitor uncertainty**
   - Use **sensors** to detect and quantify uncertainty sources at runtime.

5. **Enable self-adaptation**
   - Apply appropriate **relaxation operators** (e.g., *IN*, *AS CLOSE AS POSSIBLE TO*) to support adaptive responses.

Requirements Reflection and Mitigation Mechanisms

From Specification to Operation

- The **textual language** defines how to express relaxed requirements,
  but not *how* the system implements them.

- Operationalization requires:
  - Integration of **requirements**, **uncertainties**, and **adaptations** into one unified framework.

Requirements Reflection

- Applies **computational reflection** to requirements.
- Makes requirements **accessible at runtime** → the system can *inspect* and *analyze* them.
- Enables **runtime adaptation decisions** to mitigate uncertainties.

A Note on the Realization of Requirements Reflection

Requirements reflection ≠ adaptation goals

In self-adaptive systems, adaptation goals are runtime entities representing stakeholder requirements

Designers must translate stakeholder **requirements** into machine-readable **adaptation goals** used by the feedback loop

Realization Challenges

Full **requirements reflection**: where requirements remain *active and inspectable at runtime*

Goal modeling provides a foundation for realizing it within a **model-driven engineering approach**

The process:

- Capture uncertainties and stakeholder goals
- Progress through design, implementation, and deployment
- Keep the goal model **alive at runtime**.

Meta-Requirements for Self-Adaptation

What is the requirements problem a feedback loop for self-adaptation is intended to solve?

The **feedback loop** addresses problems related to the *runtime success or failure* of other system requirements

Thus, the **requirements of the managing system** are **meta-requirements** — i.e., requirements *about* the requirements of the managed system

These define the **concerns of the managing system** in the conceptual model of self-adaptive systems

Modeling Approach

Awareness Requirements – specify *when* adaptation is needed

- Identify situations where managed system requirements are *violated or at risk*

Evolution Requirements – specify *how* the system should adapt

- Define actions or strategies to restore or improve system behavior.

Awareness Requirements (1)

Express situations where deviations from regular system requirements are acceptable.

Define the degree of success or failure that stakeholders can tolerate.

Evaluated at runtime, enabling adaptive responses when deviations occur.

Awareness Requirements (2)

Type	Short description
Regular	A requirement that should never fail.
Aggregate	Imposes a constraint on the success/failure rate of another requirement, which can refer to a frequency, an interval, or a bound on the success/failure.
Trend	Enables comparing success/failure rates over a number of periods in order to specify how success/failure rates of a requirement evolve over time.
Delta	Enables the specification of acceptable thresholds for the fulfillment of a requirement.
Meta	An awareness requirement about another awareness requirement.

Awareness Requirements Examples

Regular AR1: Fail-safe Operation ⟶ NeverFail

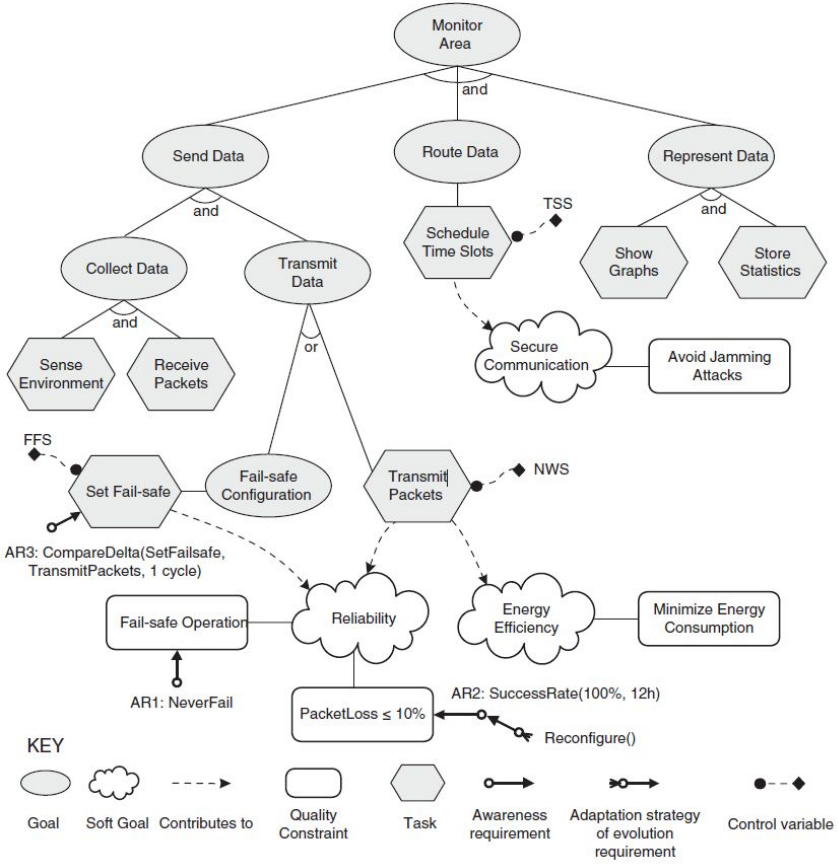
Aggregate AR2: Packet Loss ≤ 10% ⟶ SuccessRate(100%, 12h)

Delta AR3: ComparableDelta(SetFailsafe, TransmitPackets, 1 cycle)

Modeling Awareness Requirements

Goal-Oriented Requirements Engineering (GORE):

- Represents **goals**, **softgoals**, and **quality constraints**.
- Decomposes goals into tasks linked to responsible actors.
- Awareness requirements (AR1–AR3) integrated into the **goal model** to monitor runtime satisfaction.



Evolution Requirements

Prescribe the actions or changes the system must perform **when awareness requirements are triggered**.

Define **how** the system should *adapt or evolve* in response to detected deviations.

Represented as **sequences of operators** that modify:

- Elements or instances of the **goal model**, or
- The **managed system** and/or **managing system** (feedback loop)

Evolution Requirements Operators

Operator	Short description of effect
apply-config(C)	The managed system should change from its current configuration to the specified configuration *C*.
change-param(R, p, v)	The managed system should change the parameter *p* to the value *v* for all future executions of requirement *R*.
find-config(algo, ar)	The managing system should execute algorithm *algo* to find a new configuration to reconfigure the managed system when awareness requirement *ar* fails. The managing system should provide the algorithm all data that is required to find a suitable configuration.
disable(R)	The managed system should stop trying to satisfy requirement *R* from now on. If *R* is an awareness requirement, the managing system should stop evaluating it.
enable(R)	The managed system should resume trying to satisfy requirement *R* from now on. If *R* is an awareness requirement, the managing system should resume evaluating it.

An Example

```
//** c: current configuration; NWS: network settings; AG: adaptation goals */
Reconfigure(algo: bestAdaptationOption(c, NWS, AG), ar:SuccessRate(100, 12)) {
    c' = find-config(algo, ar);
    apply-config(c');
}
```

The adaptation strategy **Reconfigure** defines a sequence of **two operations** to deal with failures of the awareness requirement SuccessRate(100, 12) (i.e. AR2). The strategy takes as arguments an **algorithm algo** to find a new configuration and the **awareness requirement ar** that triggered the strategy. The algorithm uses the **current configuration c**, the **network settings NWS** (i.e. the control variables) that define the adaptation options to select a new configuration, and the **adaptation goals AG** to determine the best option. The strategy, which is executed when the awareness requirement fails, will **find a new configuration** that satisfies the goals and **adapt the managed system** accordingly.

Operationalization of Meta-Requirements

Operationalization means translating meta-requirements (awareness + evolution requirements) into executable self-adaptive behavior at runtime.

The system must include mechanisms that allow it to monitor, decide, and adapt dynamically.

Key Components Needed

Runtime Goal Model

A live representation of the system's goals, implemented (e.g., via object-oriented classes and runtime objects).

Monitoring Framework

Probes continuously track the parameters of awareness requirements (e.g., packet loss for AR2).

Mitigation Mechanisms

Implement **adaptation strategies** (from evolution requirements) to correct deviations.

System Support (Effectors)

Enact the required operations on the managed system (e.g., reconfiguration, enabling/disabling goals).

Implementation Strategy

Often realized using **Event-Condition-Action (ECA) rules**:

Event: change in monitored parameter (e.g., packet loss increases).

Condition: requirement constraint violated (e.g., loss > 10%).

Action: execute adaptation strategy (e.g., reconfigure network).

Functional Requirements of Feedback Loops

Focuses on **requirements about the behavior of feedback loops themselves**, rather than stakeholder requirements.

Ensures that **MAPE components** (Monitor, Analyze, Plan, Execute) perform their functions correctly.

1. **Software Testing**
   - Executes software against **test cases** to detect and correct errors.
   - Verifies correctness relative to a set of test scenarios.

2. **Formal Verification**
   - Uses **mathematical techniques** on a **formal model** to guarantee correctness.
   - Must be complemented by **validation** to ensure the system meets **stakeholder needs**.

Design and Verify Feedback Loop Model

Correct-by-Construction Modeling

Requires three main elements:

1. **Formal models** of MAPE-K components (Monitor, Analyzer, Planner, Executor, Knowledge).
2. **Formally specified properties** of functional requirements.
3. **Model verifier** to check that the model satisfies these properties.

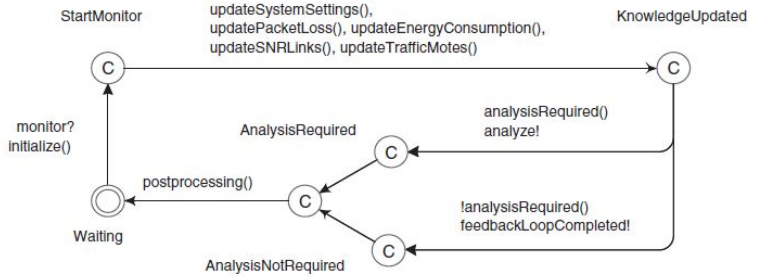
Uses **timed automata** and **Computational Tree Logic (CTL)**.

Examples of properties:

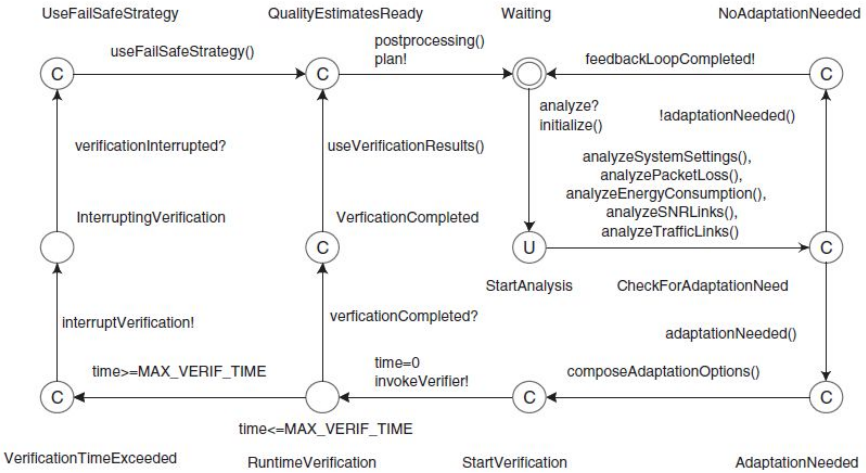
1. P1: Monitor.AnalysisRequired ⟶ Analyzer.CheckForAdaptationNeed
2. P2: Analyzer.AdaptationNeeded ⟶ Verifier.VerificationCompleted
3. P3: Analyzer.RuntimeVerification && Analyzer.time ≥ Analyzer.MAX_VERIF_TIME ⟶ Analyzer.UseFailSafeStrategy

Ensures correct behavior of **functions, guards, and invariants**.

An Example



(a) Monitor Model DeltaIoT



Monitor: updates knowledge (quality properties, uncertainties) and checks if analysis is needed.

Analyzer: evaluates if adaptation is required; composes and verifies options.

Planner & Executor

- Selects the **best adaptation option**.
- Adapts the **managed system** accordingly.

Fail-Safe Mechanism

- Analyzer enforces a **maximum verification time**.
- If exceeded → **fail-safe strategy** is applied.

Deployment Requirements

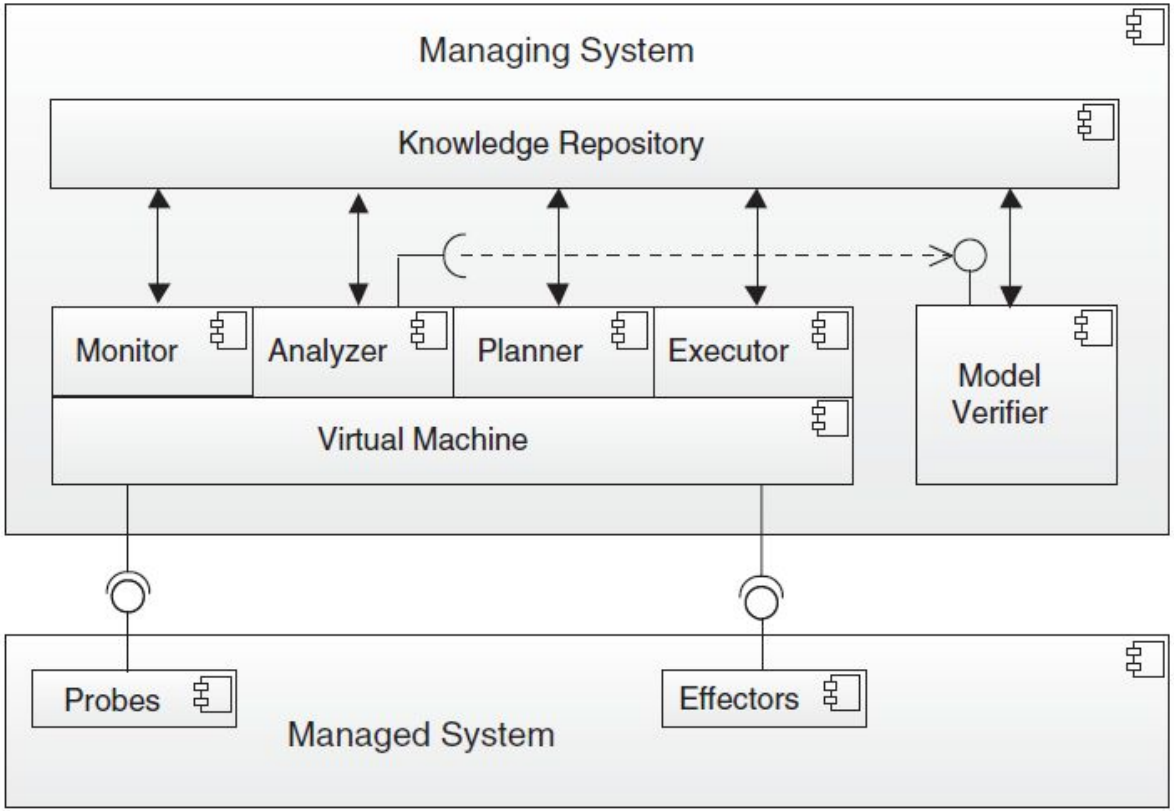
After **verification**, feedback loop models are **deployed for runtime execution** to realize self-adaptation.

Direct execution ensures that **correctness guarantees** from design and verification are preserved.

1. **Model Execution Engine**
   - Executes feedback loop models according to **model semantics**.
   - Preserves **verification guarantees**.

2. **Integration with Probes, Effectors, and Verifier** connect models to:
   - **Probes** → monitor system parameters
   - **Effectors** → enact adaptations
   - **Verifier** → analyze adaptation options at runtime

3. **Correctness of Connectors and Engine** ensured via:
   - Extensive **testing**, or
   - **Formal proofs**


Managing System
Knowledge Repository
Monitor
Analyzer
Planner
Executor
Model Verifier
Virtual Machine
Probes
Effectors
Managed System

Wave V

Guarantees Under Uncertainties

Guarantees Under Uncertainties

Focus shifts to **guaranteeing compliance with adaptation goals** despite uncertainty.

Builds on insights from:

- **Third Wave (Runtime Models):** uses probabilistic models to reason about uncertainties.
- **Fourth Wave (Requirements-Driven Adaptation):** highlights importance of uncertainty in system requirements.

Uncertainty becomes a **central driver** for self-adaptation

Uncertainties in Self-Adaptive Systems

Uncertainty in self-adaptive systems is any deviation of deterministic knowledge that may reduce the confidence of adaptation decisions made based on the knowledge.

Two main types:

- **Aleatoric:** imprecision in knowledge (common in self-adaptation)
- **Epistemic:** lack of knowledge

Poorly mitigated uncertainty can lead to **inaccurate or unreliable adaptation**, degraded quality, or **safety violations**

Managing uncertainty increases **trustworthiness** of the system

Sources of Uncertainty

System-related

- Simplifying assumptions, model drift, incompleteness, future parameter values
- Uncertain adaptation functions, decentralized decision-making, automatic learning

Goal-related

- Requirements elicitation, conflicting qualities, future goal changes

Context-related

- Inaccurate context models, noisy sensing, multiple or inconsistent data sources

Human-related

- User input variability, multiple ownership, hidden/confidential information

Sources of Uncertainty

Group	Source of uncertainty	Brief explanation
System	Simplifying assumptions	Modeling abstractions that introduce some degree of uncertainty.
	Model drift	Misalignment between elements of the system and their representations.
	Incompleteness	Some parts of the system or its model are missing and may be added at runtime.
	Future parameter values	Lack of knowledge about future values of parameters that are relevant for decision-making.
	Adaptation functions	Imperfect monitoring, decision-making, and executing functions for realizing adaptation.
	Decentralization	Lack of accurate knowledge about the system-level effects of local decision making.
	Automatic learning	Learning with imperfect and limited data, or randomness in the model and analysis.
Goals	Requirements elicitation	Elicitation of requirements is known to be problematic in practice.
	Specification of goals	Difficulty of accurately specifying the preferences of stakeholders.
	Future goal changes	Changes in goals due to new customer needs, new regulations, or new market rules.
Context	Execution context	Context model based on monitoring might not accurately determine the context and its evolution.
	Noise in sensing	Sensors/probes are not ideal devices, and they can provide (slightly) inaccurate data.
	Different information sources	Inaccuracy due to composing and integrating data originating from different sources.
Humans	Human in the loop	Human behavior is intrinsically uncertain; it can diverge from the expected behavior.
	Multiple ownership	Parts of the system provided by different stakeholders may be partly unknown.

Taming Uncertainty with Formal Techniques

Taming uncertainty focuses on the **analysis** and **planning** stages of the self-adaptive workflow. It relies on **formal techniques** applied **at runtime** to support **reliable adaptation decisions**

Decision-Making Process:

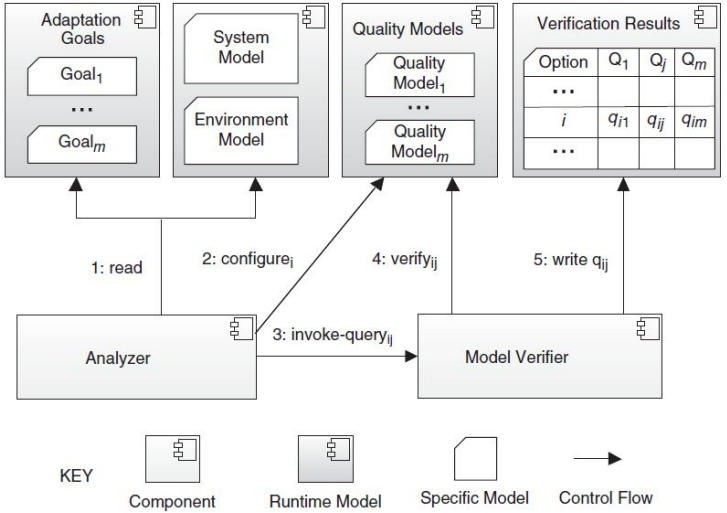
Use **formal runtime verification** to reduce uncertainty in decision-making and ensure **goal compliance** under changing conditions

1. **Analysis Phase:**
   - The **analyzer** uses **formal verification** to evaluate possible **adaptation options**
   - Ensures each option meets the system's goals and constraints

2. **Planning Phase:**
   - The **planner** compares the verified options
   - **Selects the best adaptation option** based on verification results and adaptation goals

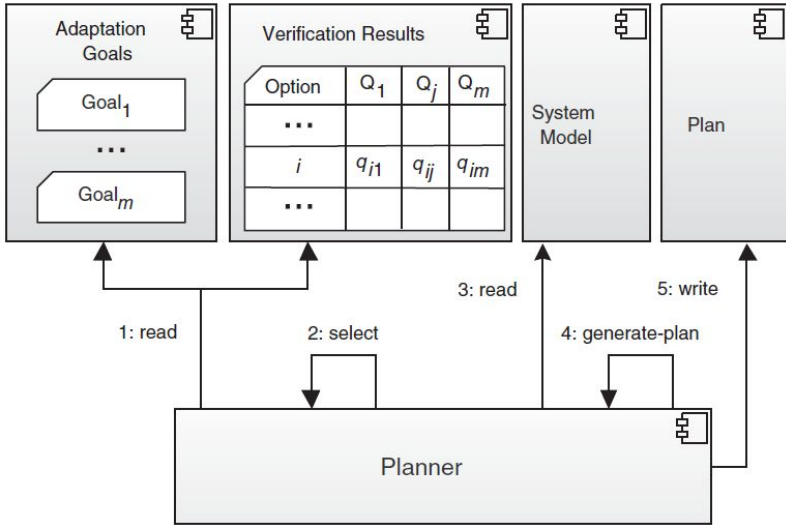
Formal Analysis Workflow

1. **Read:** The analyzer reads the **runtime models** of the **managed system**, **environment**, and **adaptation goals** to identify possible adaptation options (system reconfigurations)
2. **Configure:** For each adaptation option, the analyzer **configures the quality models**, setting parameters that represent system configurations and environmental uncertainties
3. **Invoke Query:** The analyzer calls the **model verifier** to check each configured quality model against a **property** corresponding to a specific **adaptation goal**
4. **Verify:** The **model verifier** performs **formal verification**, producing an estimate (prediction) of a **quality property** for each adaptation option
5. **Write:** The verification results are **written to the knowledge base**, recording the predicted quality values for all adaptation options



Selection of Best Adaptation Option

1. **Read:** The planner retrieves adaptation goals and verification results from the analyzer
2. **Select:** It applies a **decision-making mechanism** such as a **utility function** (weighing quality properties and their importance) or a **rule-based system** to rank and choose the best adaptation option. Options failing threshold criteria (e.g., too costly) are discarded
3. **Read:** The planner reads the **current system configuration** to prepare for adaptation
4. **Generate Plan:** A **plan** is generated to transition the system from its current state to the selected configuration
5. **Write:** The plan is **stored in the knowledge base**, where the **executor** retrieves it to carry out the adaptation actions



Exhaustive Verification to Provide Guarantees for Adaptation Goals

This approach ensures that a **self-adaptive system** meets its **adaptation goals** through **runtime quantitative (exhaustive) verification**.
Quantitative verification checks if **probabilistic or quantitative properties** (e.g., message loss probability, response time) hold for **Markov-based models** of systems with stochastic behavior

In the **MAPE feedback loop**, the **analyzer** applies this verification at runtime to evaluate the **quality of each adaptation option**, while the **planner** selects the option that best satisfies the goals

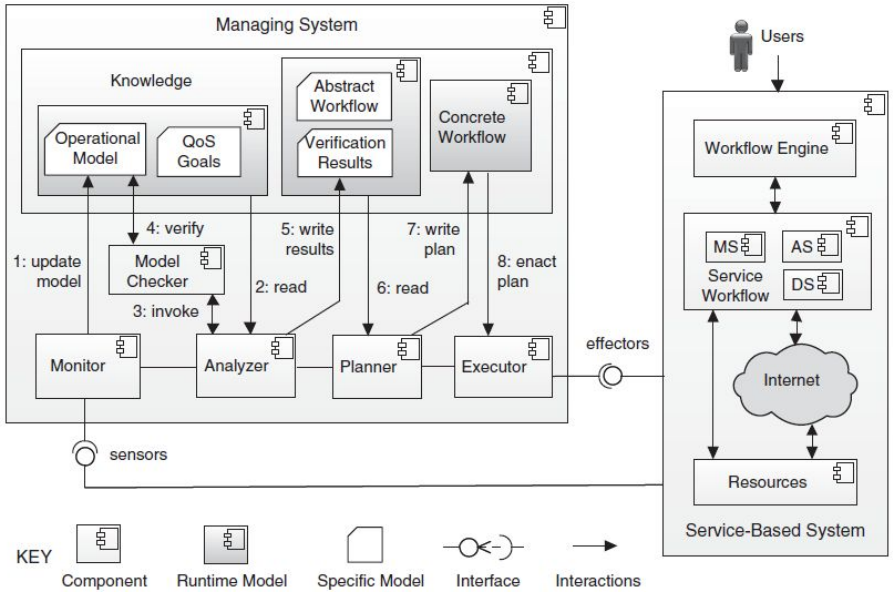
In the example system, users access **composed web services** provided by different **providers**, each offering varying **quality attributes** like **reliability, response time, and cost**. Verification ensures that the chosen configuration provides the best trade-off between these properties

Architecture of a feedback loop that uses exhaustive verification

Context: Variations in the *failure rates* of concrete services

System: Variations in the *availability* of concrete services

Human: Variations in *how users use the services*



R1: The probability that a failure of the workflow execution occurs **must be less than 0.14**

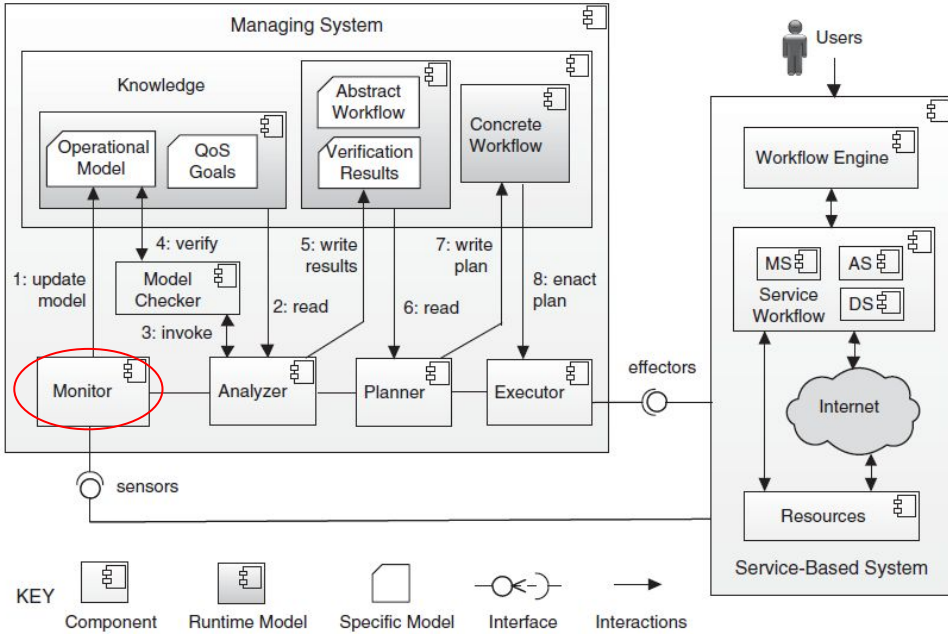
R2: The cost of executing the workflow should be **minimized**

Architecture of a feedback loop that uses exhaustive verification

The **Monitor** observes the live behavior of the service-based system, for instance, tracking how often a medical analysis service or an alarm service fails during operation

It updates an **operational model** (a probabilistic model such as a Discrete Time Markov Chain) with the current reliability and usage data of each service

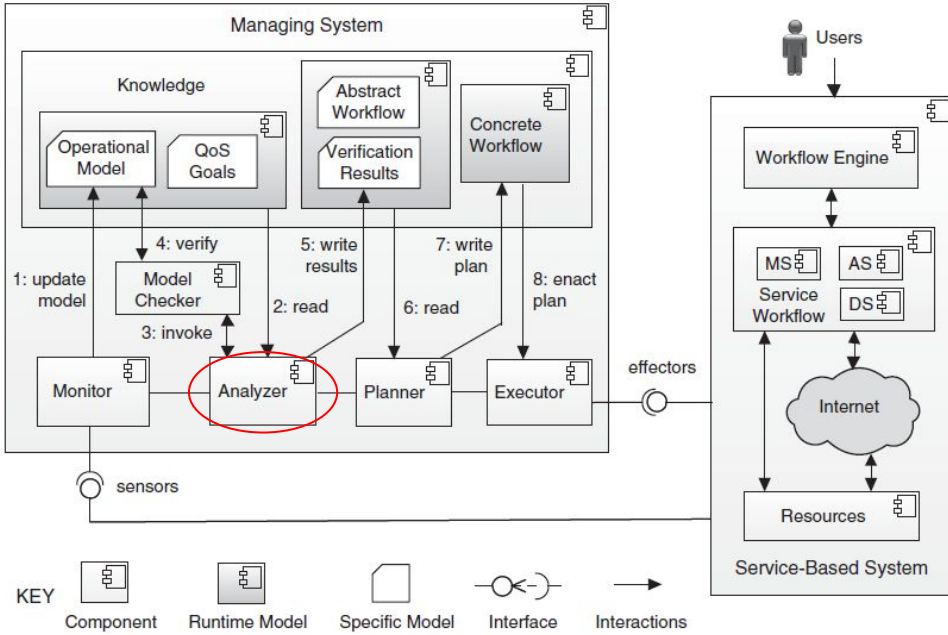
***Example**:* if the failure rate of the drug service suddenly increases, the Monitor updates that value in the runtime model so later analysis reflects the new risk



Architecture of a feedback loop that uses exhaustive verification

When quality goals (like reliability) are not met, the **Analyzer** starts an evaluation of possible configurations. It instantiates the operational model for different combinations of services and uses **quantitative verification** (e.g., via the PRISM tool) to check which ones meet the reliability requirement

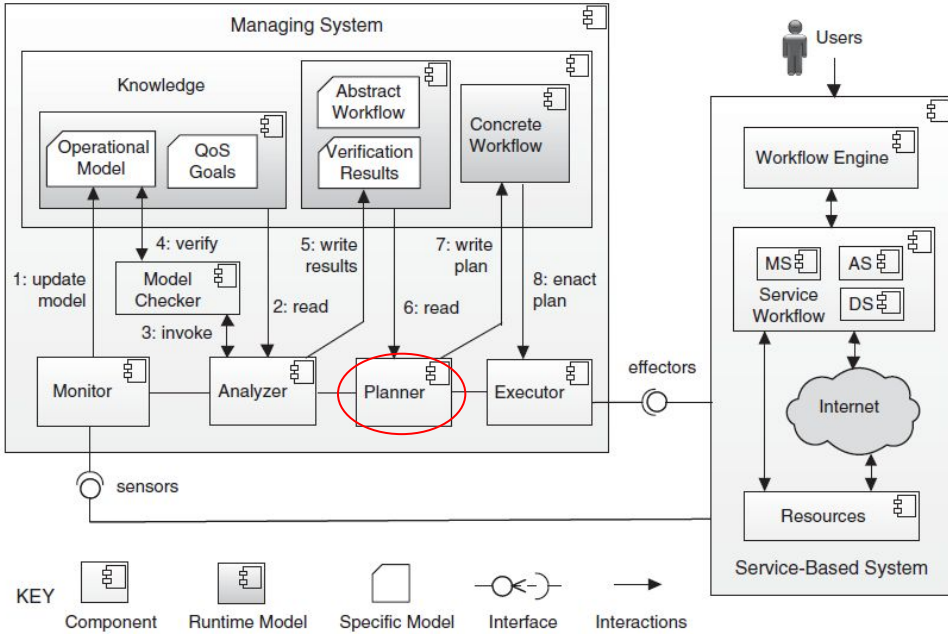
***Example**:* it may verify if using “Drug Service 2” together with “Alarm Service 1” keeps the probability of workflow failure below the accepted threshold



Architecture of a feedback loop that uses exhaustive verification

The **Planner** examines the verified configurations and selects the one that best satisfies the adaptation goals while optimizing other criteria such as cost

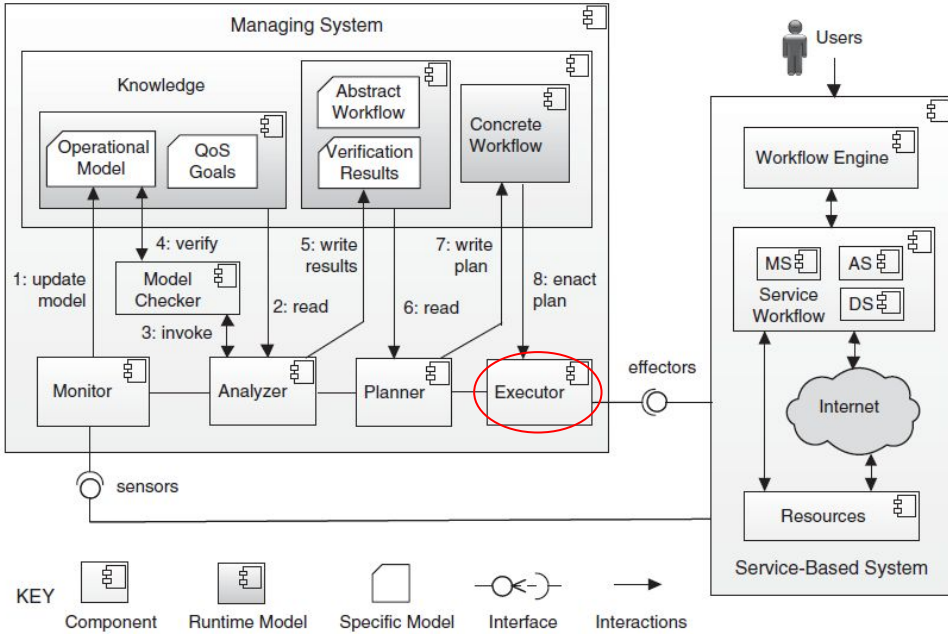
***Example**:* among all service combinations that keep reliability above target, the Planner chooses the one that provides the same reliability at the lowest cost



Architecture of a feedback loop that uses exhaustive verification

Once a plan is ready, the **Executor** applies it to the running system. It reconfigures the workflow by deactivating the old service instances and activating the new selected ones

***Example**:* it may replace the failing "Drug Service 1" with "Drug Service 2" and connect it to the workflow engine so requests are automatically redirected



Statistical Verification to Provide Guarantees for Adaptation Goals

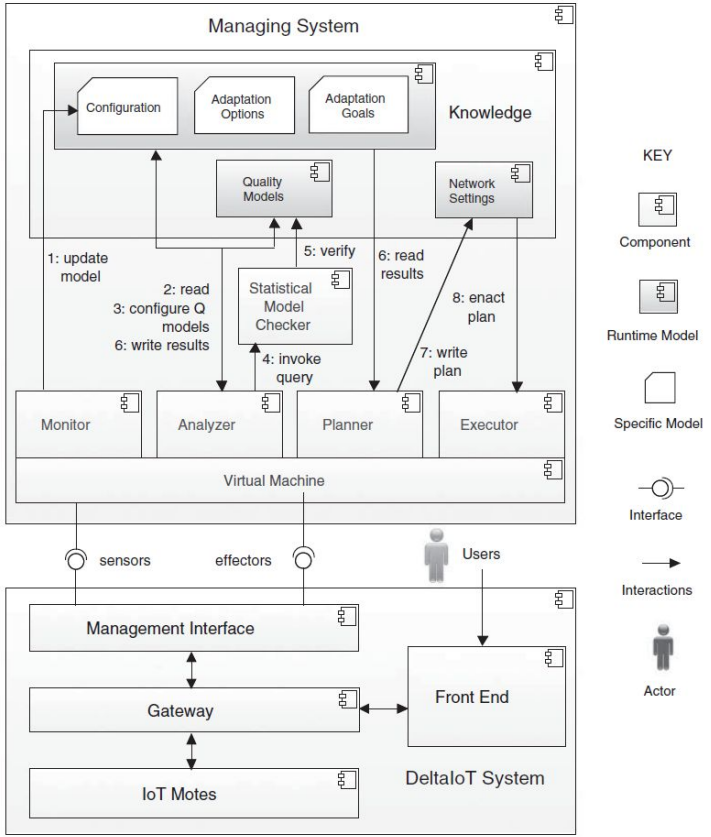
This approach ensures efficient runtime adaptation under uncertainty using **statistical model checking (SMC)**, which estimates property satisfaction through **simulation and statistical analysis** rather than exhaustive verification

It provides results with controllable **accuracy–confidence trade-offs**, reducing verification time and computational cost

An Example

Context: network interference

System: fluctuating traffic load



R1: average packet loss ≤ 10% over 12 hours

R2: minimize energy consumption

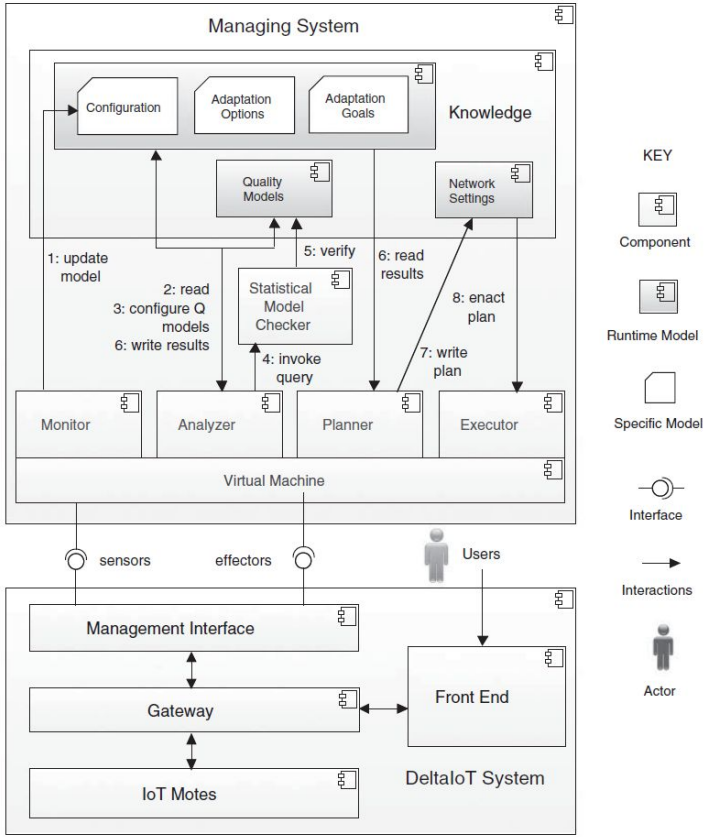
R3: use default settings if no feasible configuration exists

An Example

The **Monitor** collects data (packet loss, traffic load) to update a **Configuration model**

When variations exceed thresholds, the **Analyzer** evaluates possible configurations using **quality models** that predict energy use and packet loss

These models are verified through **UPPAAL SMC**, which runs multiple simulations to estimate average energy consumption with defined accuracy and confidence



The **Planner** then selects the best adaptation option by applying two goals:

1. Filter configurations that meet R1 (packet loss ≤ 10%).
2. Among those, choose the one with minimum energy consumption (R2).

Executor finally enacts these settings, completing the adaptation cycle.

Proactive Decision-Making using Probabilistic Model Checking

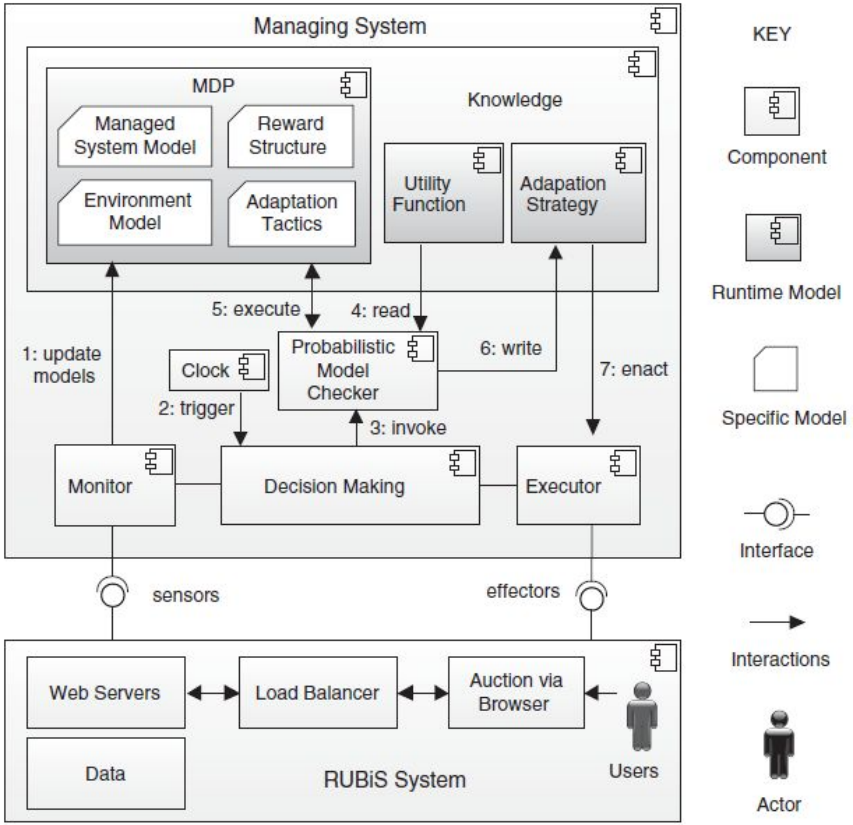
This approach enhances self-adaptation by making **proactive decisions** — anticipating future changes instead of reacting only when problems occur

It is inspired by **Model Predictive Control (MPC)** and applies **probabilistic model checking** to plan a sequence of adaptations that **maximize long-term utility** rather than short-term fixes

Example System: RUBiS (1)

RUBiS is a **web-based auction application** composed of:

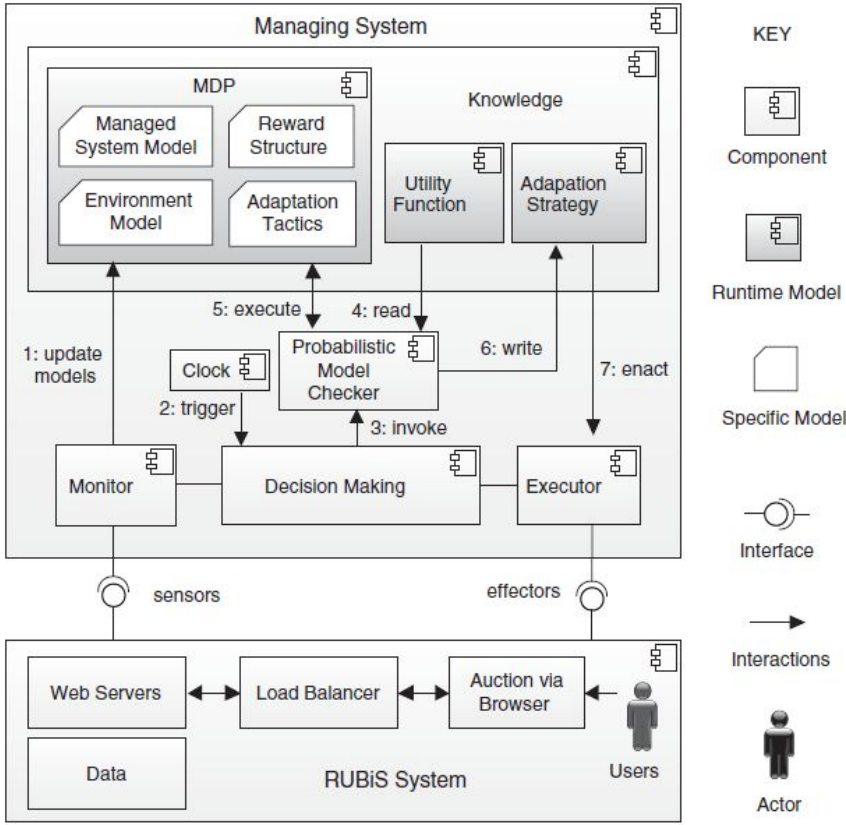
- A **Web tier** receiving user requests
- A **Server tier** processing requests
- A **Database tier** storing data
- A **load balancer** distributing requests among servers



Example System: RUBiS (2)

Response time of requests

System load and resource usage



The **goal** is to **maximize overall utility**, balancing **revenue** and **operational cost** over a look-ahead horizon

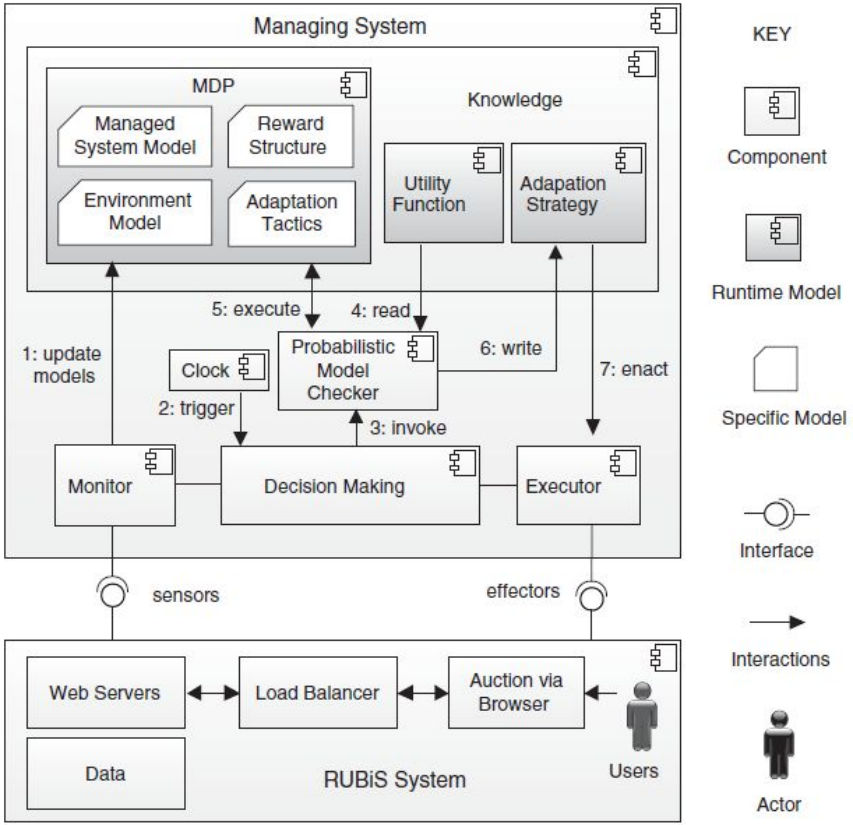
Example System: RUBiS (3)

Server management

- **Add server** (increases capacity; slow due to startup latency)
- **Remove server** (frees cost instantly)

Brownout mechanism

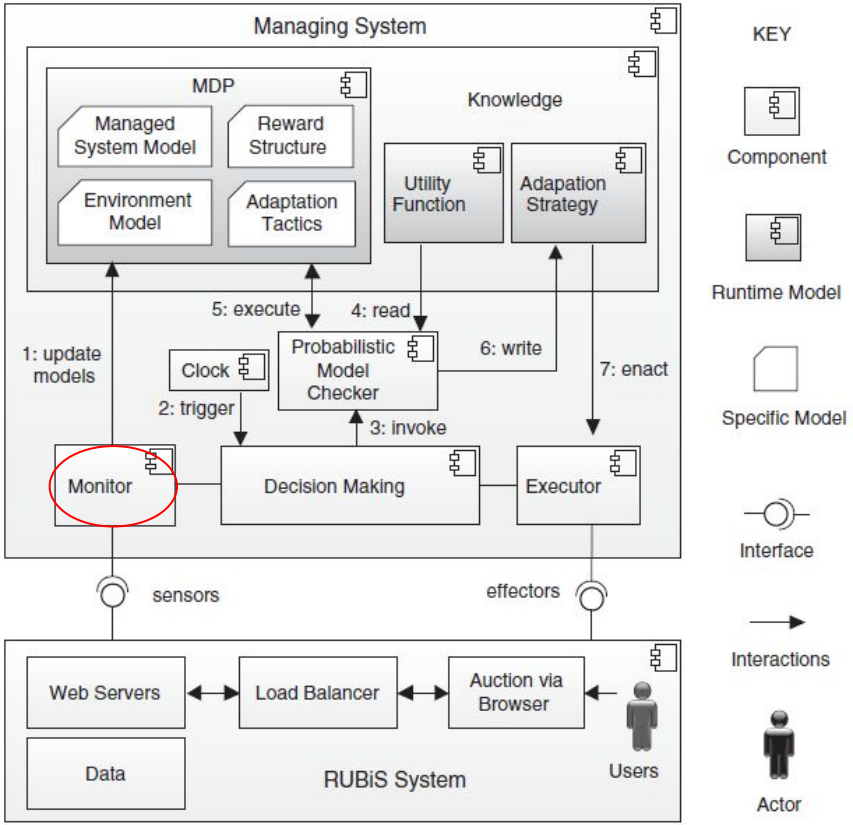
- **Dimmer control** adjusts how much *optional content* (like recommendations) is included in responses.
- Lowering the dimmer reduces load but slightly lowers user experience (and revenue).



Example System: RUBiS (4)

Monitor collects data such as:

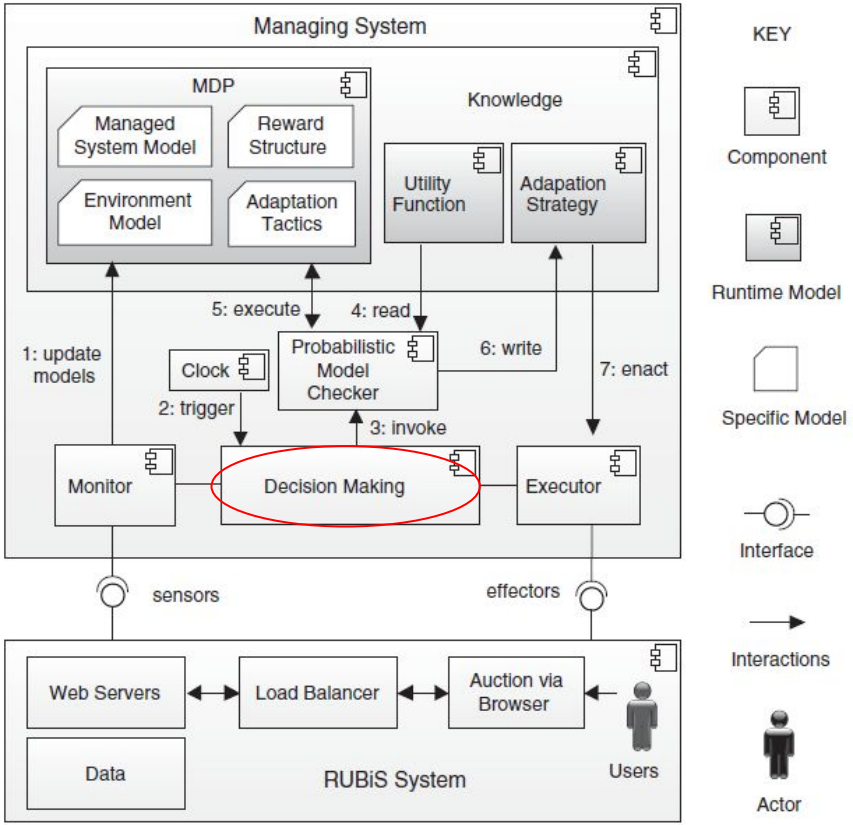
- Current number of active servers
- Current dimmer setting
- Request arrival rate
- Average response time



Example System: RUBiS (4)

Decision-Making (Analyze + Plan combined)

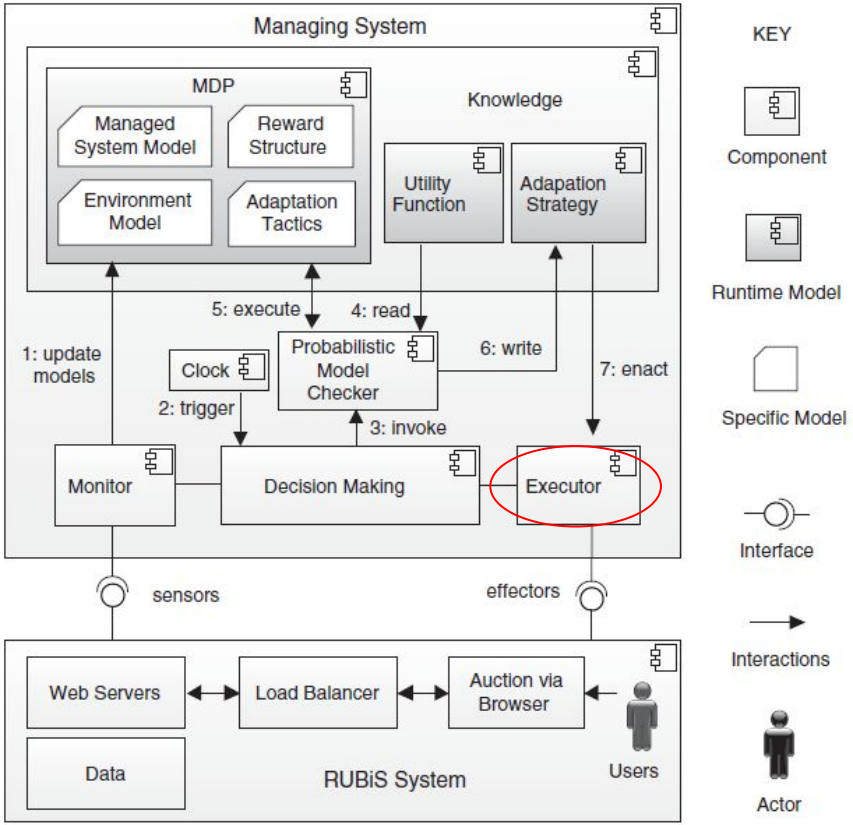
- Runs periodically (every τ minutes).
- Uses a **Markov Decision Process (MDP)** model to simulate system behavior and uncertainties (e.g., possible traffic patterns)
- Employs a **probabilistic model checker** (e.g., PRISM) to:
  - Explore possible future evolutions over a **look-ahead horizon (n·τ)**
  - Evaluate which tactics maximize **accumulated utility**
  - Synthesize an **optimal adaptation strategy**



Example System: RUBiS (5)

Applies only the **first set of actions** from the best strategy (e.g., “reduce dimmer and add one server”)

The strategy is recomputed at the next cycle, keeping the system adaptive to new conditions



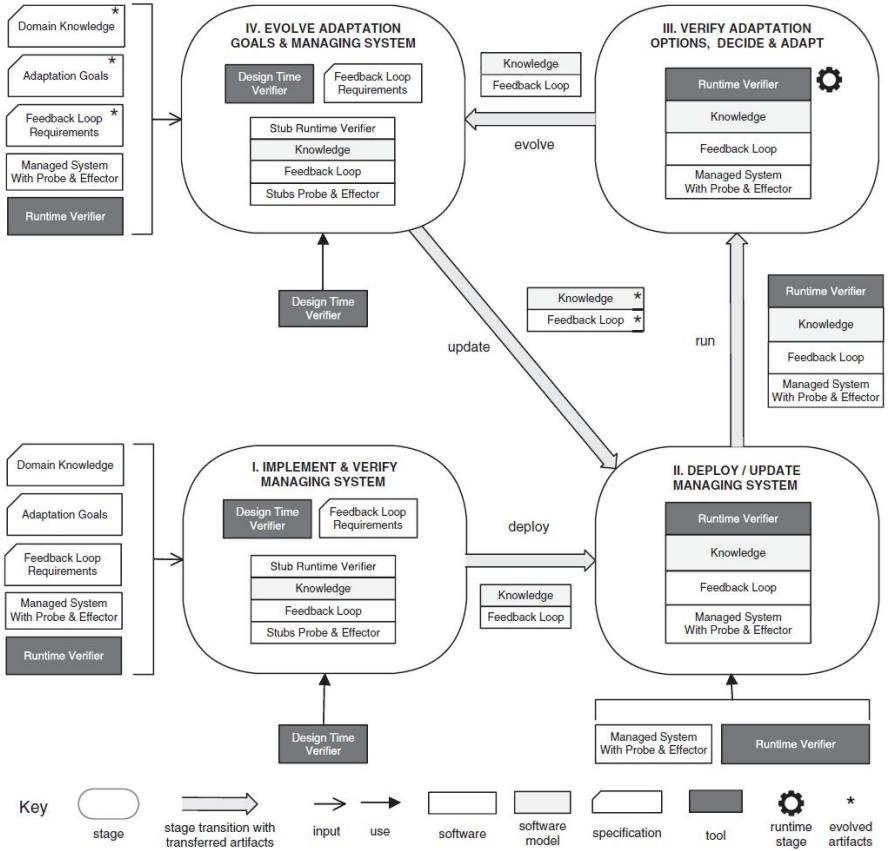
Integrated Process to Tame Uncertainty

Uncertainty prevents full assurance of requirement compliance before deployment

Self-adaptive systems address this by **collecting runtime data**, **analyzing uncertainties**, and **adapting dynamically**

An **integrated process** links **offline (human)** and **online (machine)** activities across the system life cycle

The four stages of the integrated process to tame uncertainty



Stage I – Implement and Verify the Managing System

Goal: Build and verify the feedback loop (MAPE) and runtime models

Activities:

- Encode **adaptation goals** (e.g., “keep latency < 2s”) in a **machine-readable form**
- Build **models** for the managed system, its **environment**, and **quality parameters** (e.g., response time, reliability)
- Define **uncertainty parameters** (e.g., variable network load) and **adaptation options** (e.g., add server, lower quality)
- Verify correctness with **feedback loop requirements**, such as ensuring safe fallback configurations.
- Use **stubs** and **model checking** or **testing** to validate behavior

Example: For RUBiS, verify that adding a server triggers correctly and that the system can always fall back to a single-server configuration

Stage II – Deploy the Managing System

Goal: Install and connect the verified managing system

Activities:

- Deploy the MAPE loop and **link it to probes and effectors** in the managed system
- Configure the **runtime verifier** (e.g., model checker) for evaluating adaptation options
- Ensure **reliable communication** between components (tested via dedicated code)
- Load **initial parameters** (e.g., starting load estimates, cost models)

Example: Deploy the RUBiS adaptation controller and connect it to the load balancer and server control API

Stage III – Verify Adaptation Options, Decide, and Adapt (Runtime)

Goal: Continuously monitor, analyze, and adapt the system

Steps:

1. **Compose adaptation options** (e.g., add server, dim content)
2. **Assign observed values** to uncertainty variables (e.g., current request rate)
3. **Invoke runtime verification** to predict the quality for each option
4. **Select and plan** the best option using adaptation goals
5. If no valid option is found, use a **fail-safe configuration**

Example: If RUBiS detects growing load, the system evaluates whether adding a server or dimming optional content gives the best utility before acting

Stage IV – Evolve Adaptation Goals and Managing System

Goal: Install and connect the verified managing system

Activities:

- Deploy the MAPE loop and **link it to probes and effectors** in the managed system
- Configure the **runtime verifier** (e.g., model checker) for evaluating adaptation options
- Ensure **reliable communication** between components (tested via dedicated code)
- Load **initial parameters** (e.g., starting load estimates, cost models)

Example: Deploy the RUBiS adaptation controller and connect it to the load balancer and server control API

Wave VI

Control-based Software Adaptation

Control Theory for Self-Adaptive Systems

This wave applies **control theory** to design and analyze self-adaptive software systems that must operate **under uncertainty** offering mathematically grounded guarantees for their behavior

Control theory provides techniques and tools to **design and formally analyze feedback loop systems**

A **controller** forms a **feedback loop** with a system:

1. **Monitors** a system variable (affected by disturbances)
2. **Compares** it to a reference value (goal)
3. **Generates control actions** to minimize deviation

Enables **formal analysis** of system properties such as:

- **Stability** (will the system converge to desired behavior?)
- **Accuracy** (how close to the target?)
- **Settling time** (how quickly?)

Challenges for Software Systems

Applying control theory to **software** is harder than in mechanical or industrial systems because:

- Software behavior is often **non-linear**
- **Instrumenting sensors and actuators** in software can be difficult
- **Multiple, interdependent quality goals** (performance, cost, reliability) complicate modeling
- **Mathematical complexity** of control design is often beyond typical software engineering expertise

Main Control Strategies Used

Feedback Control

Adjusts system behavior based on **measured output** vs. desired output

Deals effectively with **unknown disturbances**.
(e.g., adjusting CPU allocation based on measured latency)

Feedforward Control

Anticipates known disturbances and acts **proactively**, without relying on feedback.
(e.g., pre-scaling servers before a predicted workload spike)

Three Approaches Introduced in This Wave

Proportional-Integral (PI) Control: Handles **single goals** automatically (e.g., maintaining a target response time)

Multi-goal Control with Optimization: Extends PI control to handle **multiple, possibly conflicting objectives**

Model Predictive Control (MPC): Uses **look-ahead decision-making** to optimize adaptation over a **future horizon**, balancing several goals dynamically

A Brief Introduction to Control Theory

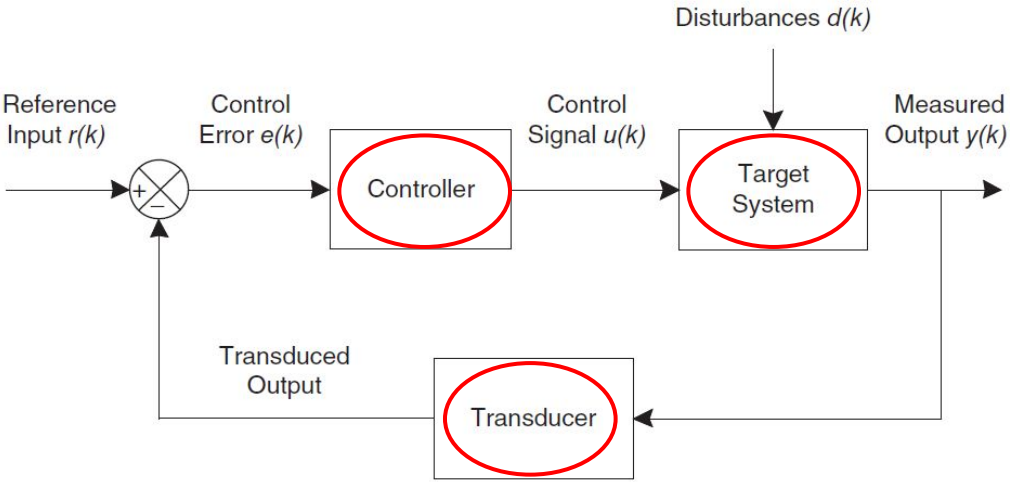
A control-based computing system consists of a **target system** that is subject to adaptation and a **controller** that implements a particular control strategy to adapt the target system

The task of the controller is to ensure that the **output** of the system is as close as possible to the **reference input**, while reducing the effects of **uncertainty** that appear as **disturbances**, **noise**, or **imperfections** in the models of the system or environment used to design the controller

Basic Elements of a Feedback Control Loop

The external element added to the target system to **dynamically adjust its behavior** based on the difference between the measured system output and the reference input

Feedback Loop



The element that **transforms the measured output** so that it can be compared with the reference input

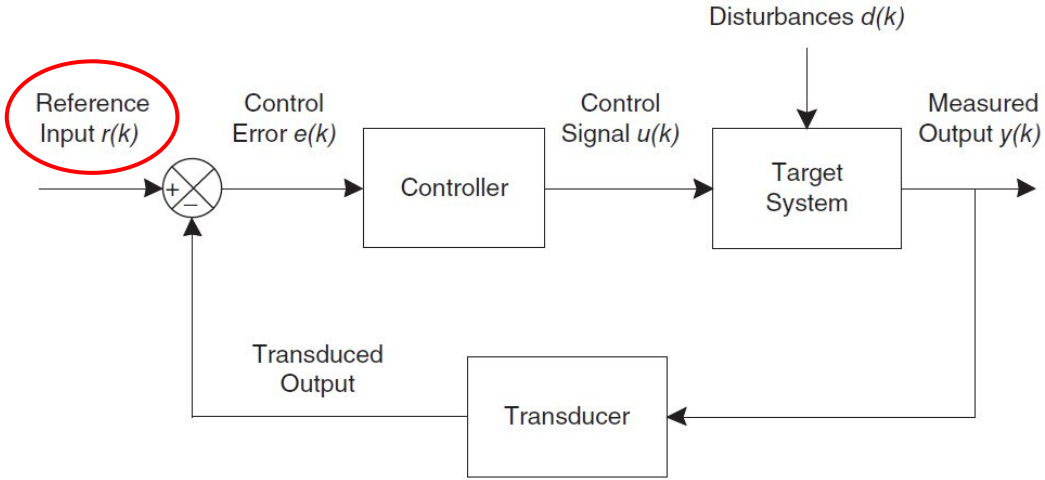
The system that is manipulated by a controller to **achieve the desired output** in the presence of disturbances

Managed System

Signals (functions of time k)

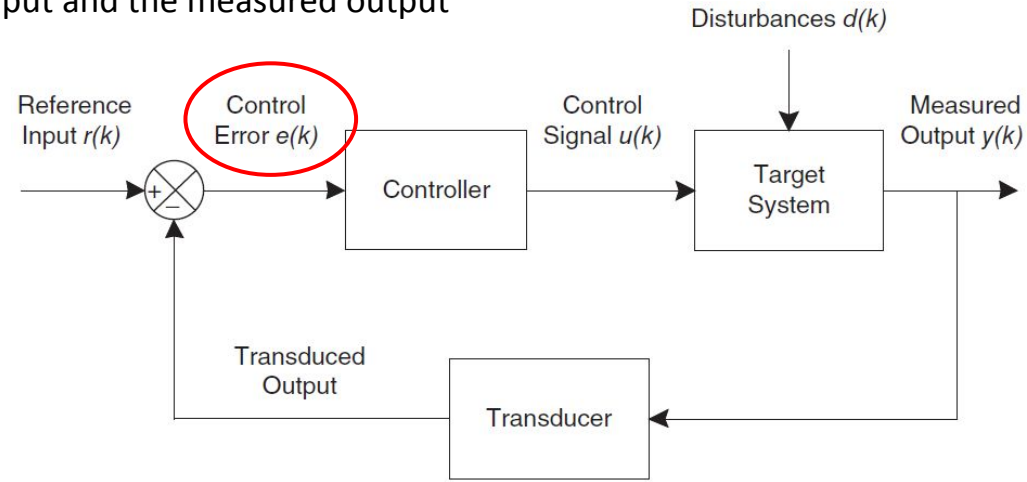
Desired value of the measured output at time k

Adaptation Goals



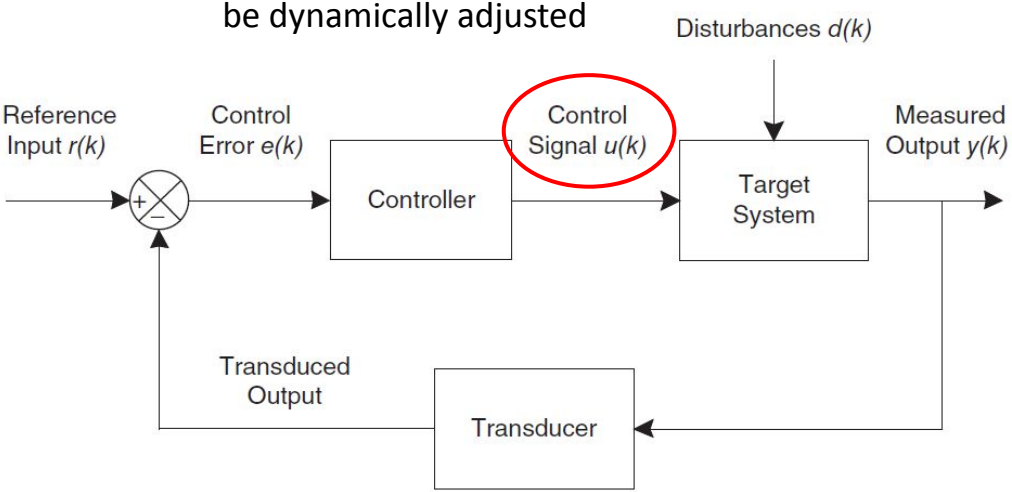
Signals (functions of time k)

Difference between the reference input and the measured output



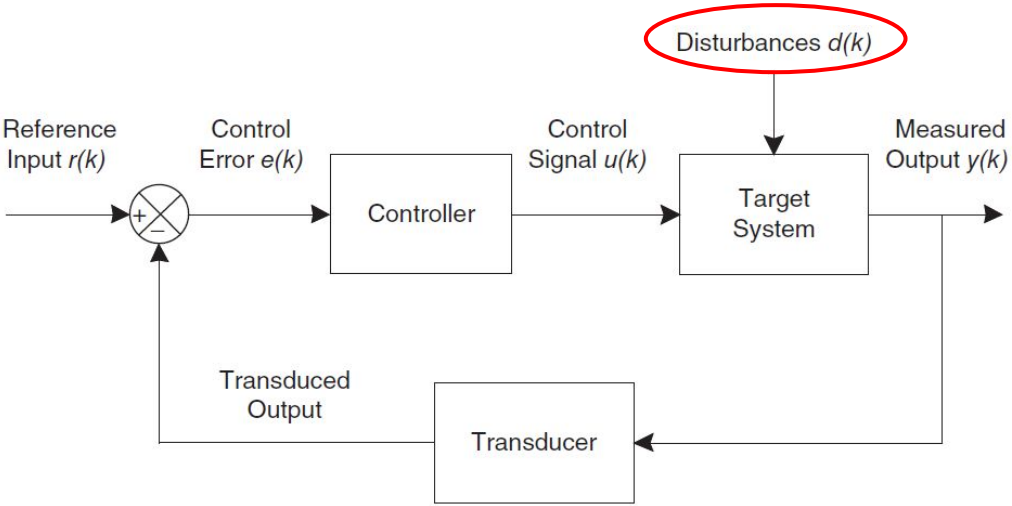
Signals (functions of time k)

Parameter setting that allows the behavior of the target system to be dynamically adjusted



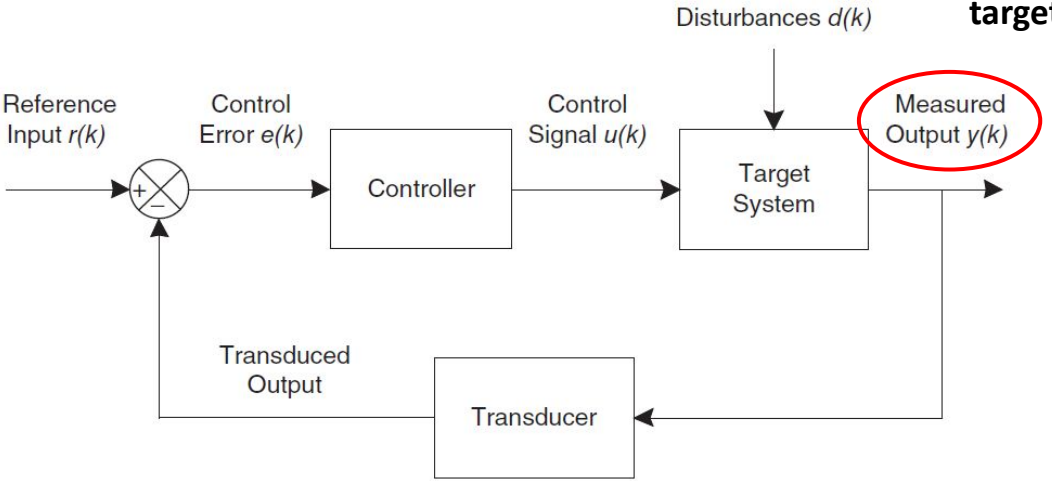
Signals (functions of time k)

Any exogenous phenomena that interfere with the effects of the control input on the measured output



Signals (functions of time k)

Measurable parameter of the target system



Purposes of Control

There are three main purposes for control:

1. **Regulatory control**: aims to keep the measured output equal to (or near) the reference input

2. **Disturbance rejection**: aims to suppress the effects of disturbances acting on the system

3. **Optimization**: aims to find the best possible value of the measured output, e.g., minimizing or maximizing a system quality property

Controller Design (1)

Using control theory to build self-adaptive software requires a **model of the target system** in the form of a dynamic system

This model defines the relationship between the **effector settings** (input variables that can be used to control the system), the **state variables** and internal dynamics, and the **control goals** (output variables)

The model of the target system is analytic and **described mathematically**. A system model can be specified manually based on knowledge of the target system, identified through experiments, or using a combination of both

Controller Design (2)

A variety of model types can be used to specify the target system, including **Markovian models, queuing models, and difference equations**. For example, in a system with a queue:

- The **input variable** could be the number of incoming requests
- The **output variable** could be the average service time
- The **state variable** could be the number of requests in the queue

To design the controller, a variety of techniques are available, depending on the information required and the guarantees they offer. One common approach is **Proportional-Integral (PI) control**, widely used in software systems

Control Properties

The main **properties** that can be analyzed in control theory are **SASO**:

- Stability
- Accuracy
- Settling time
- Overshoot

Control Properties: Stability

A system is stable if **for any bounded input, the output is also bounded**

Stability implies that the output of feedback control (**controlled variable**) **converges** to an equilibrium value after a change

Stability ensures **the system operates in a region where it performs as required**

Control Properties: Accuracy

Accuracy refers to the **convergence of the measured output to the goal** or optimal value

Accuracy **ensures control goals are met**, for example achieving required throughput without violating response time constraints

Control Properties: Settling Time

Settling time expresses **how long it takes** for the controller to reach the steady-state value

It accounts for the duration of the transient phase and is important when the system must react to sudden environmental changes or changes in goals

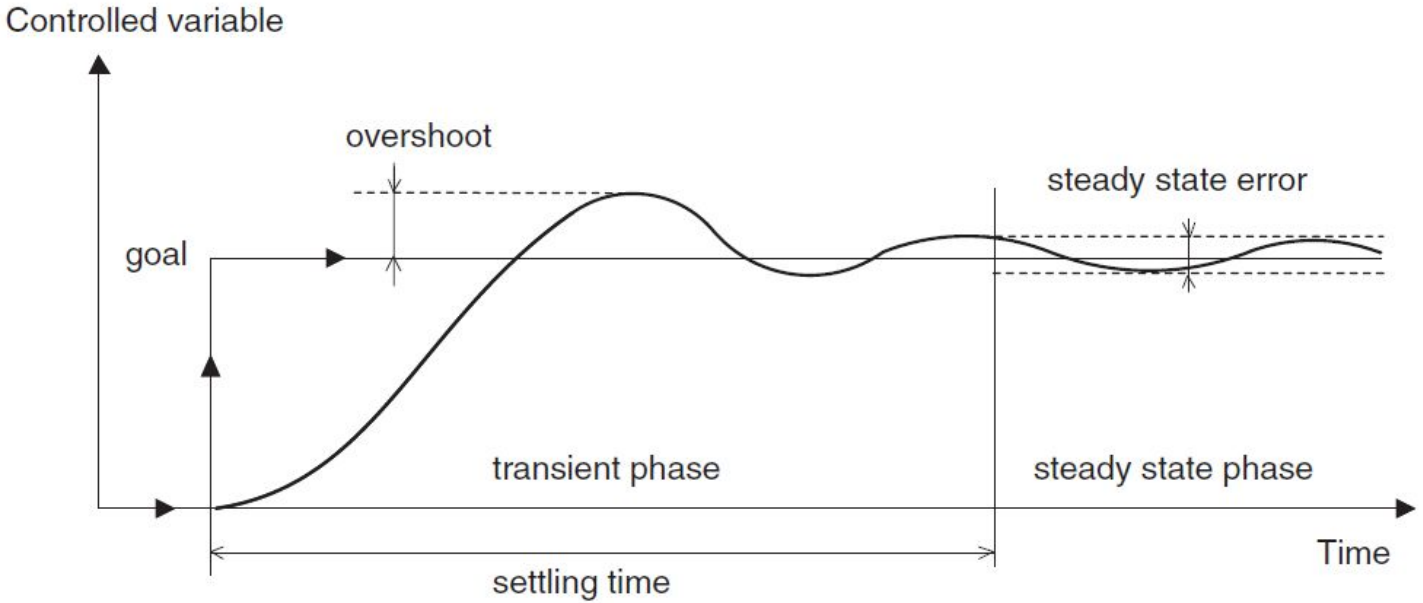
Fast settling time is desirable but may make the system follow noise or transient variations

Control Properties: Overshoot

Overshoot is the **maximum difference** between the **measured value** and the **goal** during the transient phase

Limited overshoot is preferred, as high overshoot increases variability and may temporarily violate system requirements

Control Properties: an Overview



SISO and MIMO Control Systems

The basic control system structure considers a single instance of each element, but in practice, multiple instances may be used

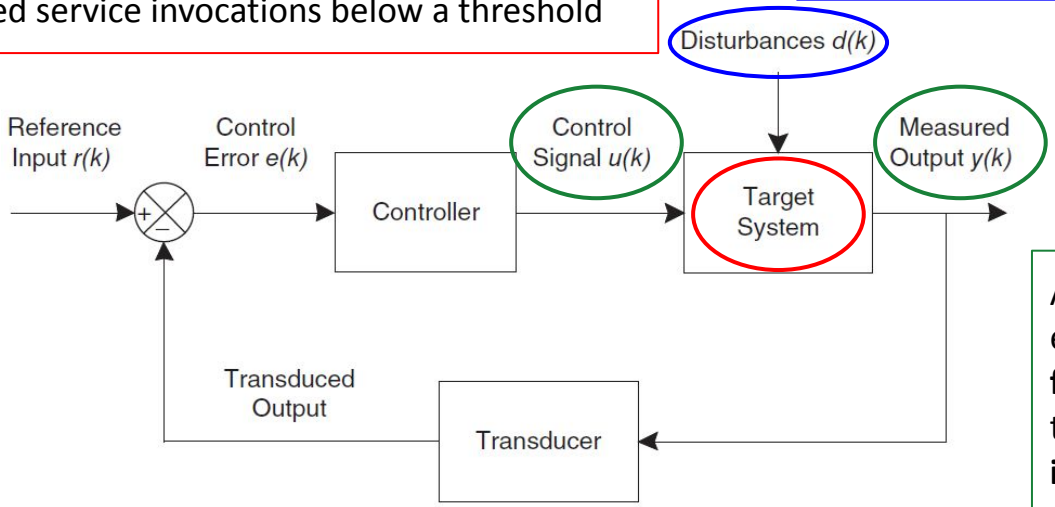
Single-Input Single-Output (SISO) control system: one output is controlled by one control signal

Multiple-Input Multiple-Output (MIMO) control systems may be used, where multiple outputs are controlled by multiple control signals

SISO Example

Web service system wants to keep the number of failed service invocations below a threshold

Disturbances include **arrival rates** and types of service requests, which change dynamically

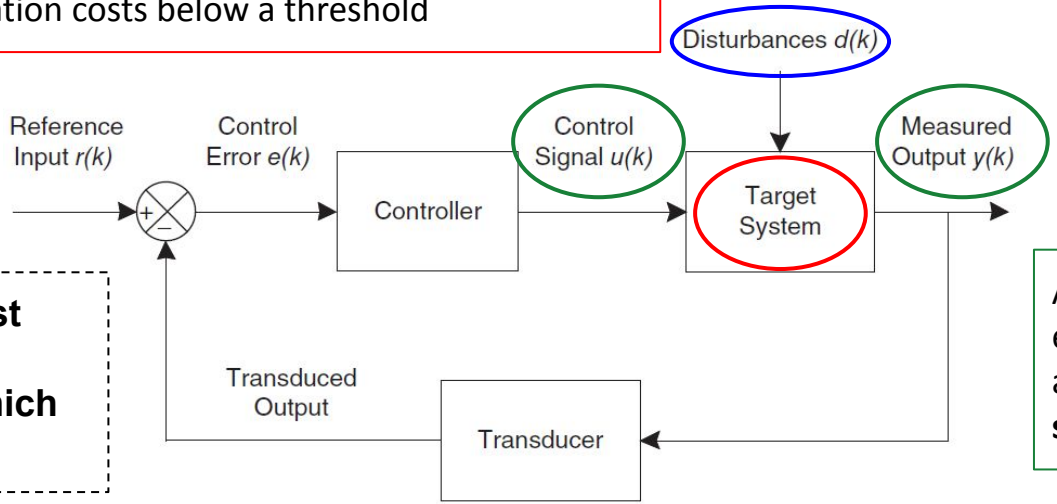


A controller is added to each server to track **failure rates** and control the selection of **service instances**

MIMO Example

Web service system wants ALSO to keep service invocation costs below a threshold

Disturbances include **arrival rates** and types of service requests, which change dynamically



Now, the controller must ensure both low failure rates and low costs, which may conflict

A controller is added to each server to track **cost** and adjust the **service selection**

Adaptive Control

Adaptive control adds an additional control loop that adjusts the controller itself

It is used to cope with **slowly occurring changes** in the system or to compensate for **inaccuracies in the initial system model**

Adjustments are made based on **measurements collected at runtime**

Automatic Construction of SISO (1)

These approaches rely on the following assumptions:

1. The **requirements** that need to be satisfied by the control system are known and can be **translated into quantifiable goals**
2. The **target software system** is available and can be used to **run experiments**
3. The target system provides a set of **sensors** to measure whether the goals are satisfied
4. The target system provides a set of **actuators** to modify the system to realize the goals

Automatic Construction of SISO (2)

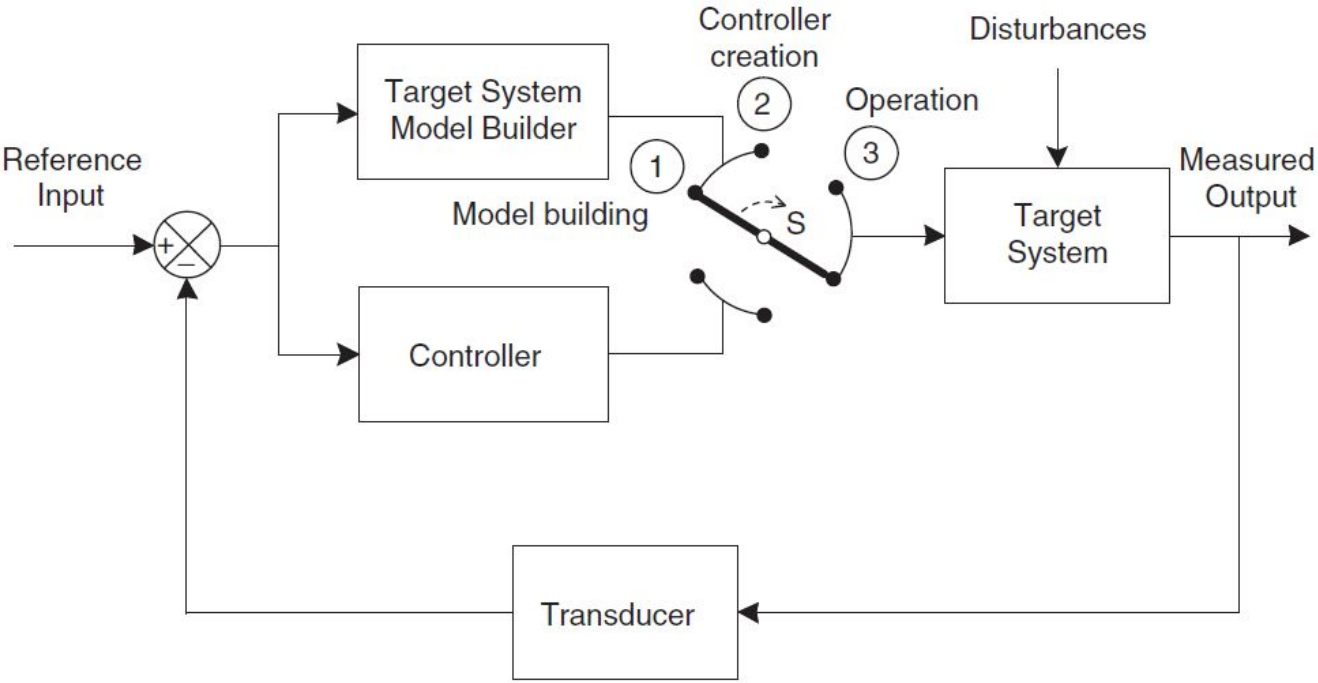
Under these assumptions, an **approximate model** of the software system can be built automatically

This model can then be used to **construct a controller** that ensures the target system achieves its goals

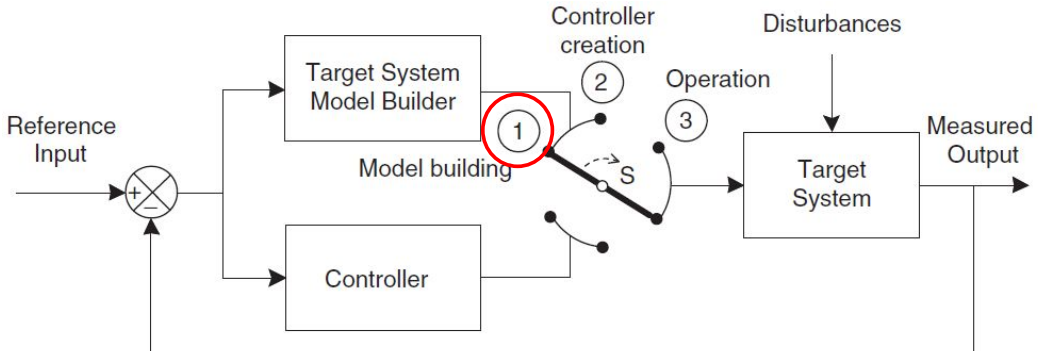
A basic approach for a **single goal and single actuator** is based on the **Push-Button Methodology**, which is a pioneering method for automating controller construction for self-adaptive software

Despite its simplicity, this approach **provides formal guarantees** for the dynamic behavior of the system

Phases of Controller Construction



Phases of Controller Construction (1)

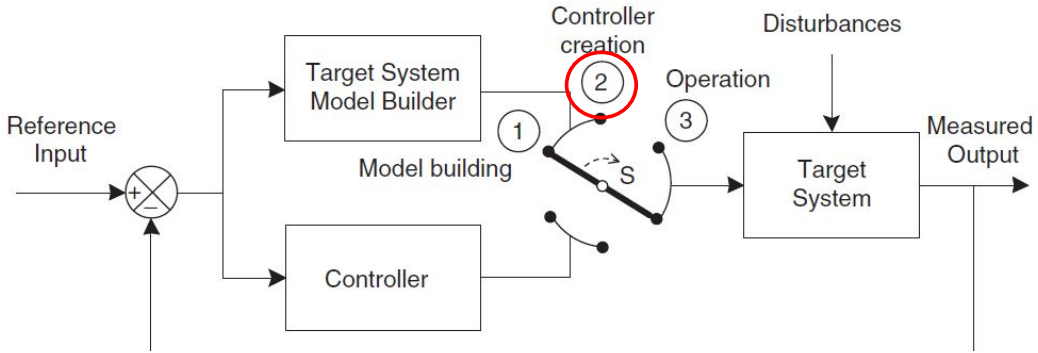


Model Building Phase: This phase **automatically constructs a linear model** of the target system

The model is **identified by running on-the-fly experiments** on the software of the target system and testing it using a set of systematically sampled values of the **control signal**, and the effects on the **measured output** are observed

By analyzing the results, a **system model** is generated that captures how changes in the control signal affect the output

Phases of Controller Construction (2)



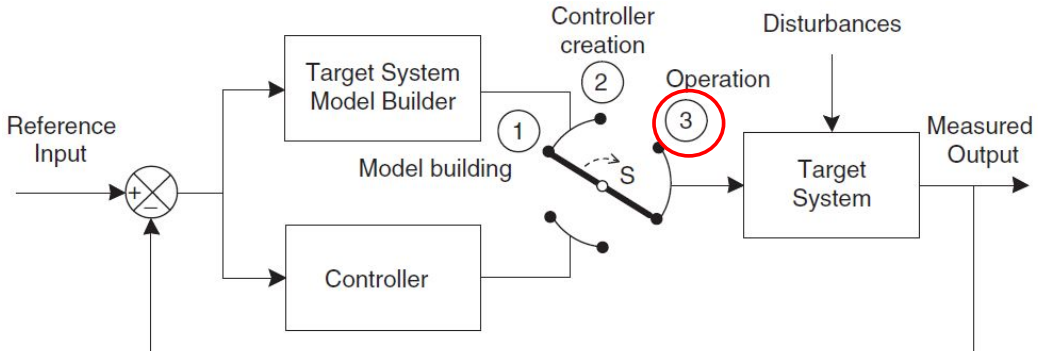
Controller Creation Phase: The results from the model building phase are used to **automatically create a controller**

The goal of the controller is to **keep the output close to the reference input** while rejecting disturbances

The controller selects a control signal based on the **previous value** and the **difference between the desired reference value and the measured output**

The controller uses the information from the system model to determine the appropriate adjustments

Phases of Controller Construction (3)



Operation Phase: In this phase, the **controller exercises control** on the target system

Its purpose is to **keep the output as close as possible to the reference input** during runtime, compensating for disturbances and changes in the system

Phases of Controller Construction

S**ingle adaptation goal and a single control signal** is often too restrictive for real-world applications

For example, users of a **geo-location service** may not only want the service to be reliable, but also to have **fast response times**, while ensuring that the **cost of using the service** does not exceed an agreed fee. Such requirements call for an approach that can automatically construct a controller to satisfy **multiple goals**, often including an **optimization goal**

Approach emerged to realize a **control schema with multiple inputs that control multiple outputs** and can handle multiple goals.

Model Creation Phase

In this phase, the system automatically builds a **set of linear models** of the controlled system, one for each adaptation goal

Each model represents the relationship between a control input (what the controller can change) and a measured output (what is observed from the system)

To build each model, the system performs **on-the-fly experiments**: it systematically varies the control inputs and records the corresponding changes in the outputs

Controller Creation Phase

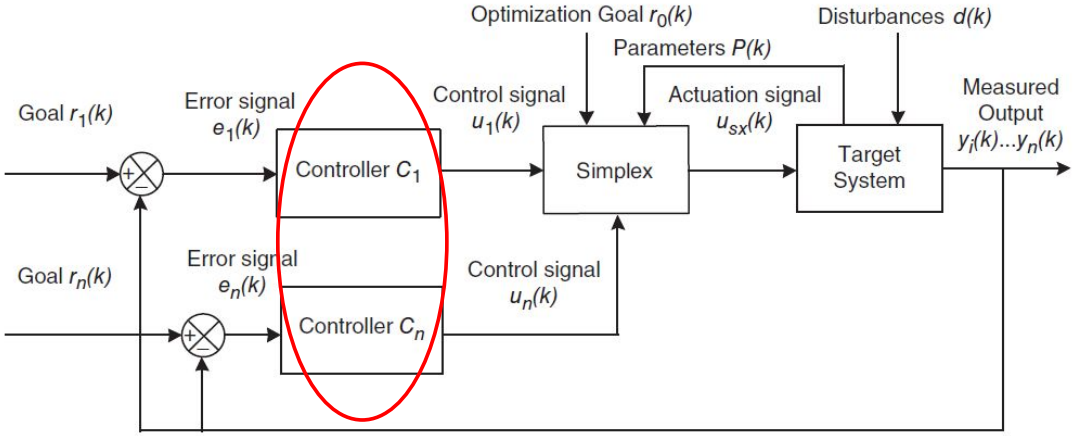
Once the models are built, the system automatically creates **one controller for each goal**
Each controller is responsible for keeping its corresponding system property (e.g., reliability, performance, cost) close to its desired value, even when the environment changes.

The controller uses:

- The **model** that describes how its control actions influence the system
- The **error** between the desired value and the measured outcome
- A **parameter** that determines how strongly it reacts to deviations (its responsiveness)

In essence, each controller continuously monitors how well the system meets its goal and adjusts its control signal to reduce the difference between the desired and observed behavior

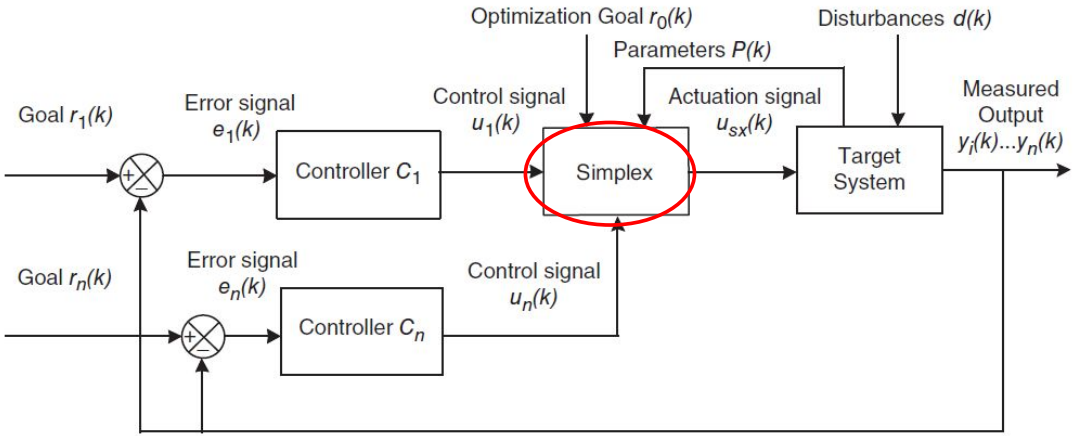
Operation Phase (1)



Each controller runs in parallel, producing control signals that guide the system toward its goals while rejecting disturbances. To handle changes over time, controllers can update their models automatically

Operation Phase (2)

Because multiple goals can sometimes conflict an **optimization mechanism** is used to balance them



This is achieved through the **Simplex method** technique which receives all control signals, analyzes system parameters and constraints, and produces a final **actuation signal**

→ a coordinated adjustment of the system that best satisfies reliability, performance, and cost requirements together

Wave VII

Learning from Experience

Learning Under Uncertainty

The **seventh wave** of self-adaptive systems focuses on the use of **machine learning techniques** to enhance how a system manages itself at runtime

As systems became larger and more complex, it became increasingly difficult to analyze every possible adaptation option within the short time available for making adaptation decisions. For example, in systems with **very large adaptation spaces**, performing full formal verification is often infeasible

In these situations, **machine learning** can be used to narrow the adaptation space to the most relevant options, improving efficiency while maintaining effective adaptation

Machine Learning

It is a branch of **artificial intelligence** concerned with systems that improve their performance through experience.
In simple terms, a program *learns* if its performance on a specific task improves as it gains experience.

Modern advances in computational power have made it possible to apply powerful learning algorithms in practice, especially for managing complex software systems.

At its core, a machine learning algorithm:

- **Builds a model** from data collected through observation or experimentation.
- **Uses the model** to make predictions or guide decisions about new, unseen situations.

Because data are always limited and future conditions uncertain, machine learning usually provides **probabilistic**, rather than absolute, guarantees

Types of Machine Learning

1) Supervised Learning

The algorithm learns from data that includes both **inputs and expected outputs.** Through repeated optimization, it learns to predict outputs for new inputs.

Example: A classifier that assigns class labels (e.g., "failure" or "success") to system behaviors based on historical data

2) Unsupervised Learning

The algorithm learns from **unlabeled data**, finding hidden structures or patterns

*Example: A **neural network** grouping similar inputs together without prior labeling*

3) Reinforcement Learning

An **agent** interacts with an environment by performing actions and receiving rewards. Over time, it learns a **policy** on how to act to maximize its cumulative reward

Example: where the system learns which actions yield the best outcomes in different situations

Machine Learning and Uncertainty

Uncertainty is central to machine learning. Algorithms learn from **imperfect or incomplete data**, discovering patterns that help make better predictions or adaptation decisions. As the system continues to operate and gather new data, its models and decisions improve over time

In **self-adaptive software**, machine learning enables the **managing system** to:

- Interpret uncertain and dynamic data from the environment
- Identify trends or anomalies
- Support **runtime adaptation decisions** that help the system meet its goals

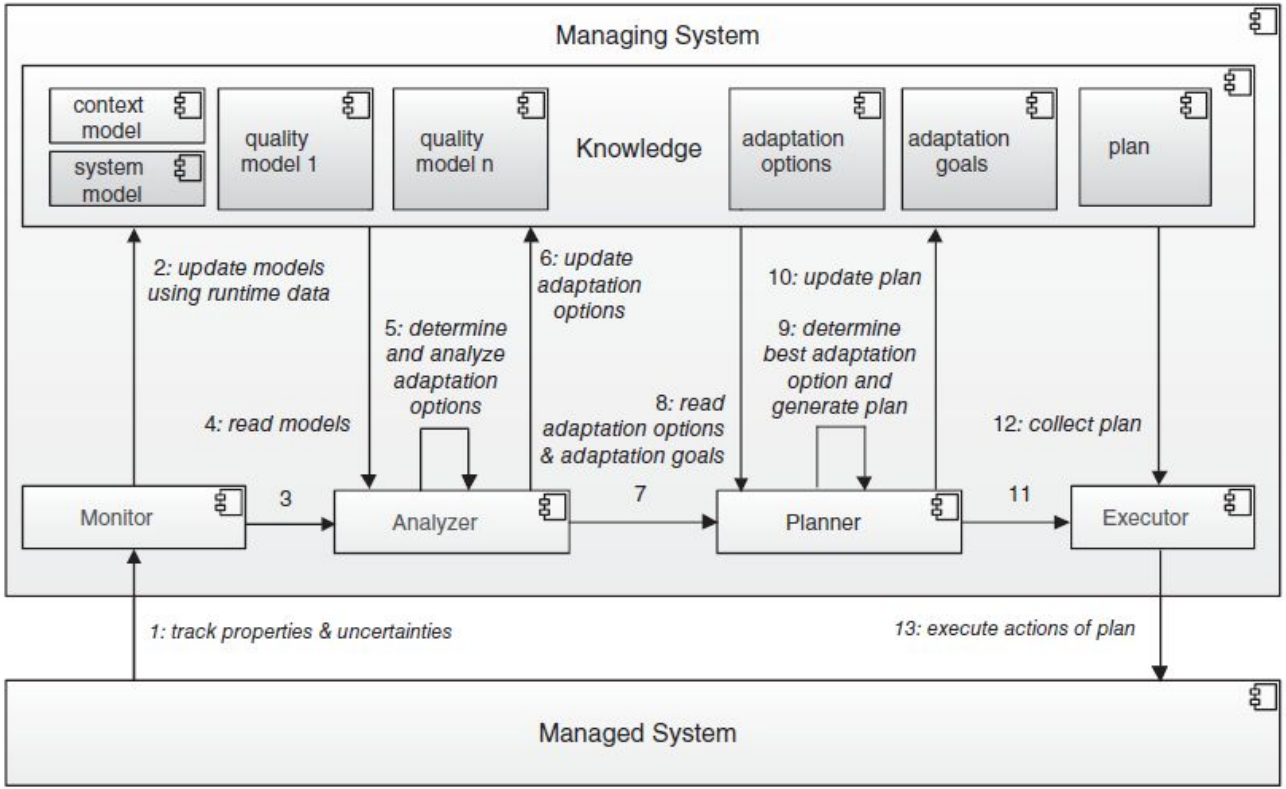
Machine Learning in Self-Adaptive Systems

In SAS, machine learning can be integrated in different MAPE activities:

1. **Learning in Monitoring:**
   The monitor function is enhanced with a **Bayesian estimator** that keeps the runtime model up to date.
   → Helps deal with **parametric uncertainty** (how system parameters evolve over time).

2. **Learning in Analysis:**
   The analyzer function is supported by a **classifier** that reduces large adaptation spaces at runtime.
   → Improves **efficiency** by focusing only on promising adaptation options.

3. **Learning Across Functions:**
   Several functions of the feedback loop use a combination of **fuzzy control** and **fuzzy Q-learning** to dynamically adjust **auto-scaling rules** in Cloud environments.
   → Supports **decision making under complex uncertainty** while continuously improving performance.


Managing System
context model
system model
quality model 1
quality model n
Knowledge
adaptation options
adaptation goals
plan
2: update models using runtime data
4: read models
5: determine and analyze adaptation options
6: update adaptation options
8: read adaptation options & adaptation goals
9: determine best adaptation option and generate plan
10: update plan
12: collect plan
Monitor
3
Analyzer
7
Planner
11
Executor
1: track properties & uncertainties
13: execute actions of plan
Managed System

Keeping Runtime Models Up-to-Date Using Learning

Runtime models are central, they are used to keep track of uncertain and changing operating conditions, analyze adaptation options, and decide how to adapt the managed system to achieve its goals

Uncertainties are encoded in runtime models as **model parameters**. These values can be determined before deployment through measurements, experiments, data analysis, or expert knowledge ⟶ however, such values are only estimates and may be inaccurate

Runtime models must remain **up-to-date** and accurately represent the actual conditions of the system and its environment ⟶ the main challenge is therefore to ensure that the models continuously reflect reality

Keeping Runtime Models Up-to-Date Using Learning

A systematic approach to address this challenge is **Bayesian estimation**, which allows model parameters to be improved as new data arrives. This method combines prior knowledge about model parameters with new observations to update their values dynamically

The approach focuses on **runtime models of quality properties** formally specified, for example, as Markov models, queuing networks, or stochastic timed automata. Such models can be analyzed at runtime using formal verification techniques.

This Bayesian approach is based on **KAMI (Keep Alive Models with Implementations)**, a pioneering method for keeping quality models up-to-date during system operation, ensuring that self-adaptive systems maintain accurate and reliable models for effective decision making.

Bayesian Approach

The **Bayesian approach** can be applied to various **probabilistic models**

Here, the focus is on **Discrete Time Markov Chains (DTMCs)**, which describe the behavior of a system through *states* and *transitions* that represent probabilities of moving between states

Key characteristics

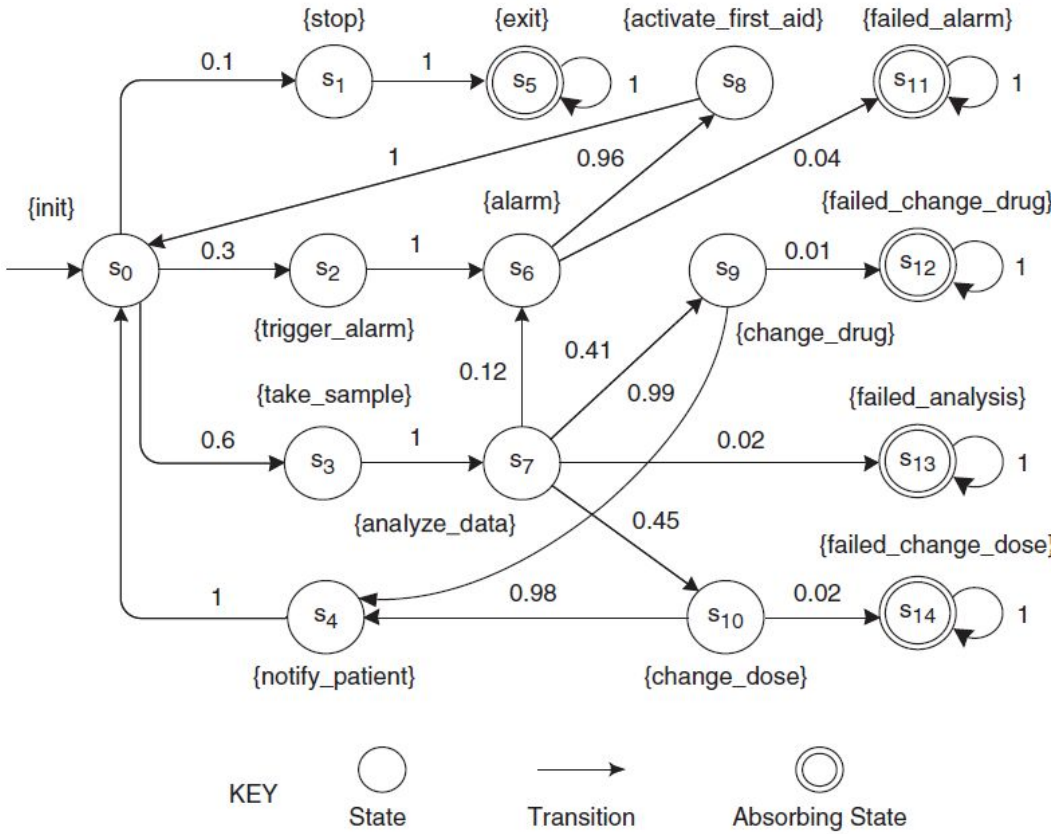
- **States (S):** represent possible situations or configurations of the system.
- **Transitions:** describe how the system can move from one state to another with a certain probability
- **Labels:** associate states with relevant properties or events (e.g., “service available” or “service failed”)
- **Markov property:** the next state depends only on the current state, not on past history

Service-based health assistance system

The system workflow is represented as a DTMC with discrete states corresponding to the different **operational steps of the health service** (e.g., user interaction, alarm triggering, service response)

Each transition between states carries a **probability value** that represents, for example, the likelihood of a service succeeding or failing

Initial probability estimates are usually provided by **domain experts** or derived from **data offered by service providers**



The **Bayesian approach** continuously tracks system and environmental elements that cause uncertainty, transforms collected runtime data into updated model parameter values, and keeps the **runtime quality model** accurate and current

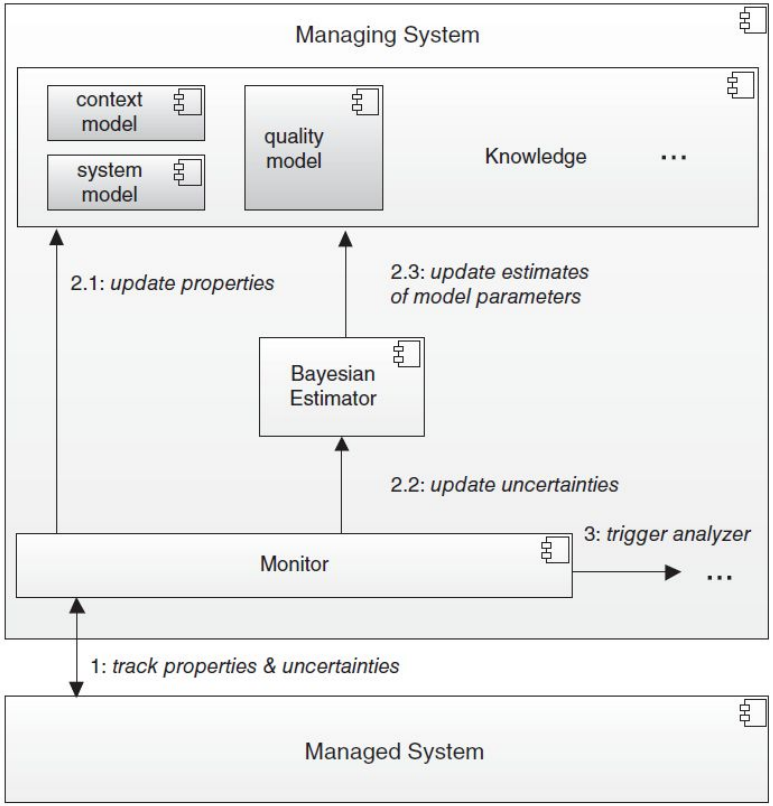
Integration in the feedback loop

The **monitor component** is equipped with a **Bayesian estimator**, which updates the model parameters as part of the feedback cycle.

This refinement divides monitoring into:

- **2.1:** Updates that do **not** require learning (e.g., deterministic parameters)
- **2.2–2.3:** Updates for **uncertainty-related parameters**, handled through Bayesian learning

Focus is on updating **quality models represented as DTMCs** (Discrete Time Markov Chains)



Reducing Large Adaptation Spaces Using Learning

The **Analyzer** in a self-adaptive system must evaluate many **adaptation options**, however:

- **Exhaustive analysis** (e.g., via formal verification) is often **too slow** for real-time adaptation
- Large adaptation spaces make **timely decision-making** infeasible

Machine Learning Solution

Introduce a **Learning Module** into the **MAPE-K feedback loop**

The learning module **reduces the adaptation space** by selecting **relevant subsets** of adaptation options

Based on **classification**, a supervised learning approach that:

- Assigns adaptation options to **classes** based on compliance with adaptation goals
- **Predicts** which options are likely to meet all goals

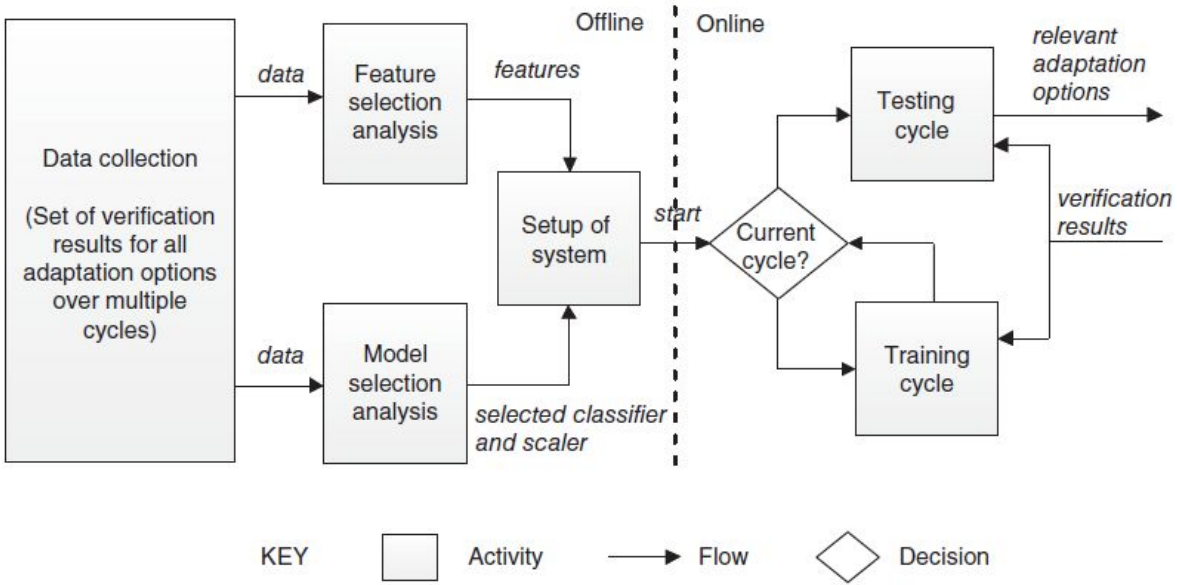
Learning-Based Feedback Loop

Relevant options (predicted to meet goals) ⟶ analyzed by the verifier

Irrelevant options ⟶ mostly skipped, but a small sample is re-analyzed periodically to detect new viable configurations

Enables **faster analysis** without sacrificing **adaptation quality**

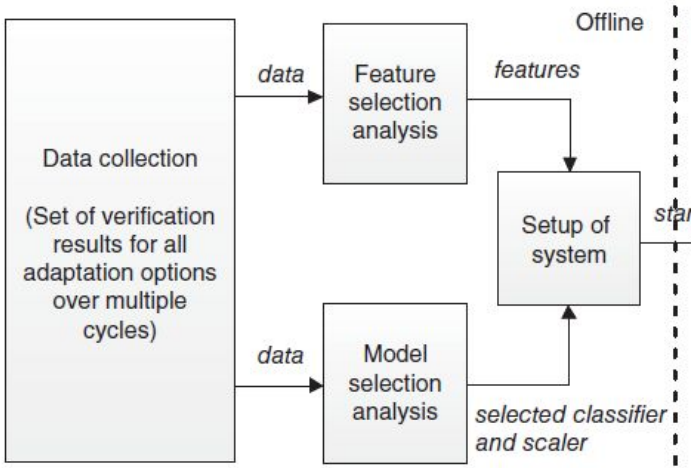
Learning Workflow



Learning Workflow: offline

Offline Phase

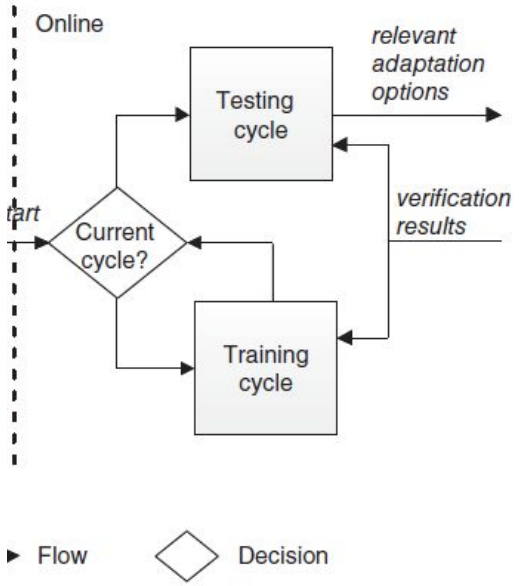
1. **Data collection** – gather verification results and system features
2. **Feature selection** – identify key features influencing performance
3. **Model selection** – choose the best classifier and scaling method based on accuracy and F1-score



Learning Workflow: online

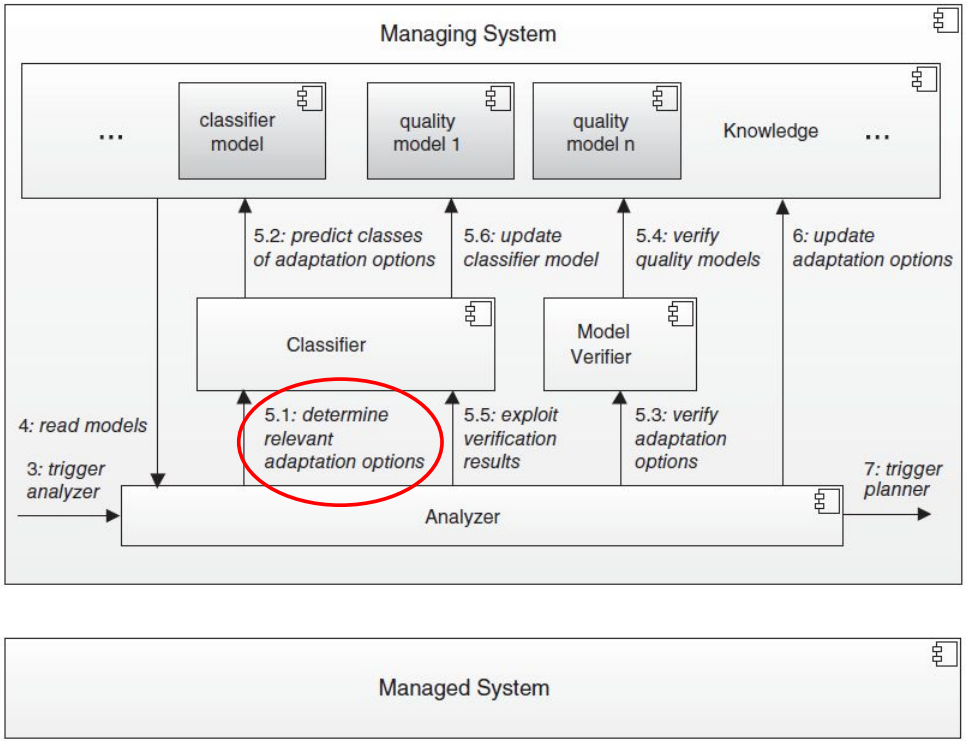
Online Phase

1. Use the classifier to **predict relevant options**
2. Perform **verification only** on those options
3. **Update** the classifier with new verification data (incremental learning)



Integrated Feedback Loop Workflow (1)

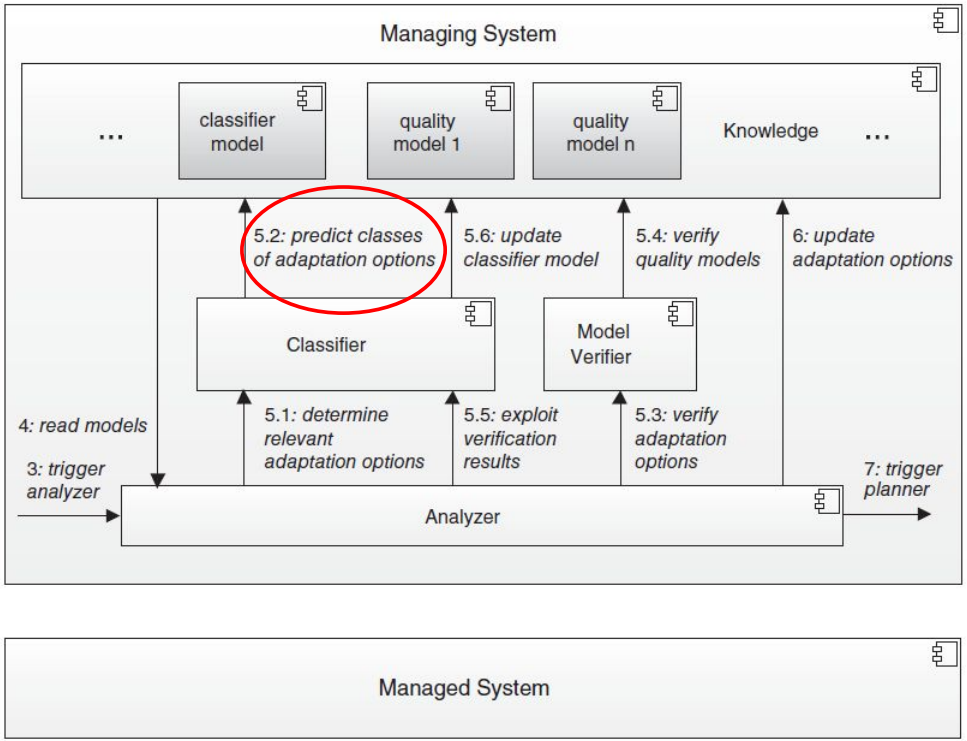
The Analyzer reads the current runtime models and invokes the classifier. The system state, environmental data, and uncertainty parameters are provided as input



Integrated Feedback Loop Workflow (2)

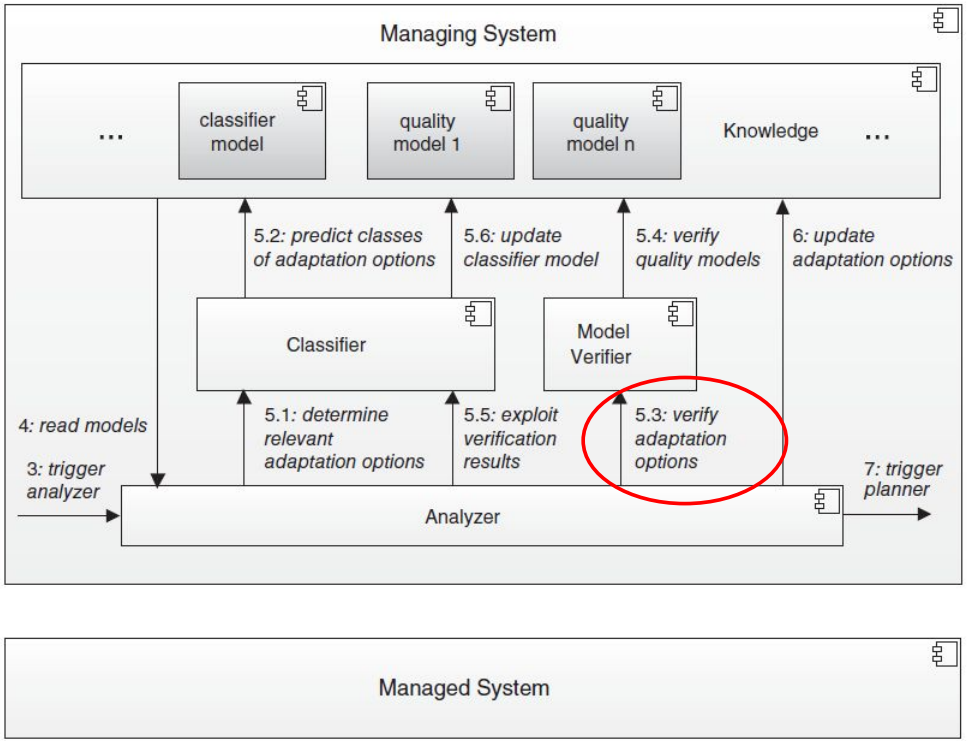
The Classifier predicts which adaptation options are most likely to meet the goals

It labels them as **relevant** or **irrelevant** based on its model. To maintain adaptability, a small random sample of “irrelevant” options is also included for exploration



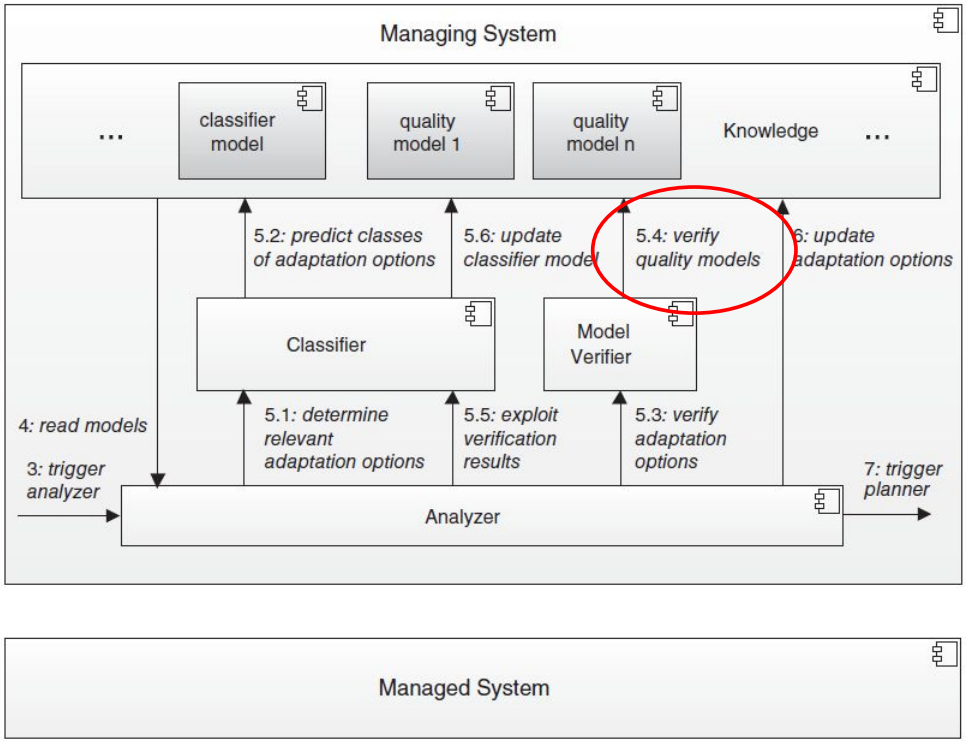
Integrated Feedback Loop Workflow (3)

The Analyzer sends the **relevant subset** of adaptation options to the Model Verifier for analysis. This reduces computational effort by focusing only on promising configurations



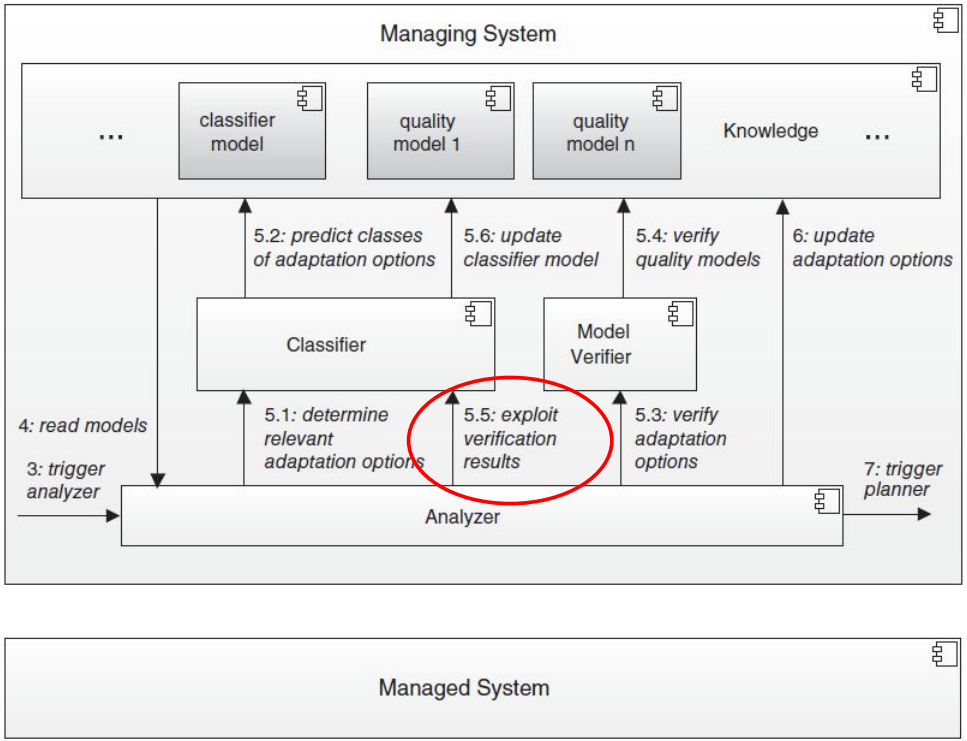
Integrated Feedback Loop Workflow (4)

The Verifier evaluates each selected option using its quality models and determines performance indicators (e.g., reliability, delay, energy usage)



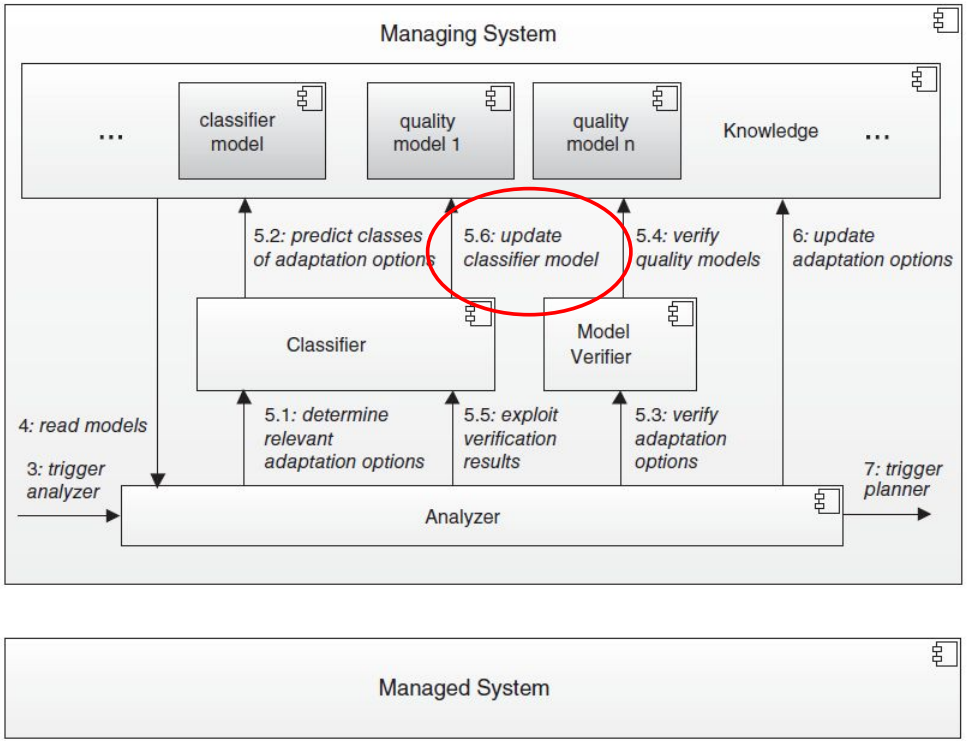
Integrated Feedback Loop Workflow (4)

Once verification is complete, the Analyzer sends the new analysis results to the Classifier. These verified results serve as **fresh labeled data** for learning

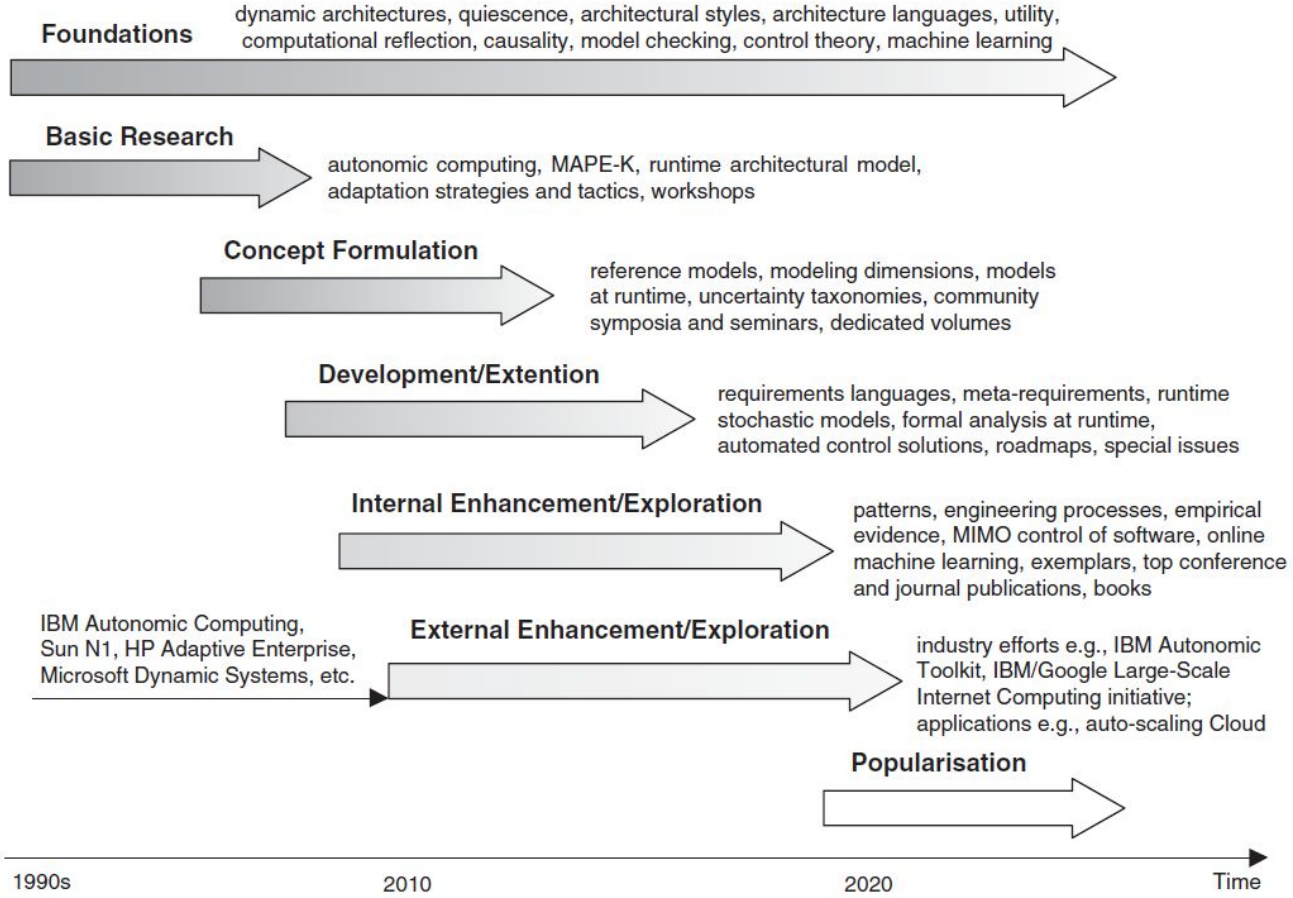


Integrated Feedback Loop Workflow (4)

The Classifier **refines its internal model** using the new verification data. This continuous update allows it to **adapt** to changing environmental conditions and evolving system behavior




The End


Foundations
dynamic architectures, quiescence, architectural styles, architecture languages, utility, computational reflection, causality, model checking, control theory, machine learning
Basic Research
autonomic computing, MAPE-K, runtime architectural model, adaptation strategies and tactics, workshops
Concept Formulation
reference models, modeling dimensions, models at runtime, uncertainty taxonomies, community symposia and seminars, dedicated volumes
Development/Extention
requirements languages, meta-requirements, runtime stochastic models, formal analysis at runtime, automated control solutions, roadmaps, special issues
Internal Enhancement/Exploration
patterns, engineering processes, empirical evidence, MIMO control of software, online machine learning, exemplars, top conference and journal publications, books
IBM Autonomic Computing, Sun N1, HP Adaptive Enterprise, Microsoft Dynamic Systems, etc.
External Enhancement/Exploration
industry efforts e.g., IBM Autonomic Toolkit, IBM/Google Large-Scale Internet Computing initiative; applications e.g., auto-scaling Cloud
Popularisation
1990s
2010
2020
Time