TOR VERGATA
UNIVERSITÀ DEGLI STUDI DI ROMA

Macroarea di Ingegneria
Dipartimento di Ingegneria Civile e Ingegneria Informatica

Self-adaptive Distributed Systems

Corso di Sistemi Distribuiti e Cloud Computing

A.A. 2026/27

Valeria Cardellini

Laurea Magistrale in Ingegneria Informatica

Self-adaptive software systems

"Intelligence is the ability to adapt to changes"

S. Hawking

- Self-adaptive (or autonomic) software system: a software system capable of dynamically adapting its operations at runtime with respect to its own internal state and environmental changes:
- Self-adaptive systems in many domains:
  - Cloud computing
  - Edge computing
  - Compute continuum
  - HPC
  - Cyber-physical systems

Self-adaptive software systems

- **Autonomic computing**: computing paradigm designed to manage the complexity and heterogeneity of IT systems through automatic adaptations
  - Inspired by the human autonomic nervous system, which controls vital functions (e.g., heart rate, body temperature) masking complexity from conscious human control
- Key capabilities of self-adaptive (or autonomic) software systems:
  - Autonomously manage functionalities and goals, with little or no human intervention
  - Adapt to changes and uncertainty within both the system itself and its environment

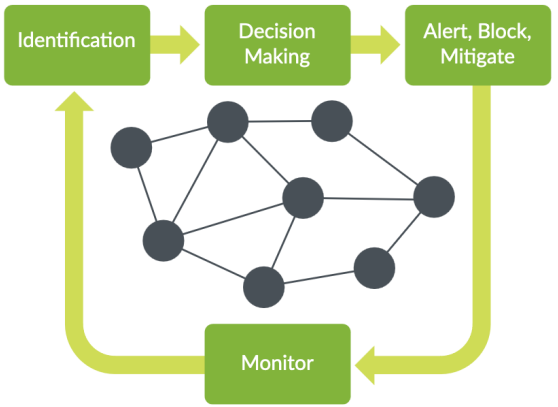
Kephart and Chess, The vision of Autonomic Computing, *IEEE Computer*, 2003

No AI/ML Processing

Self-adaptation is everywhere

kubernetes

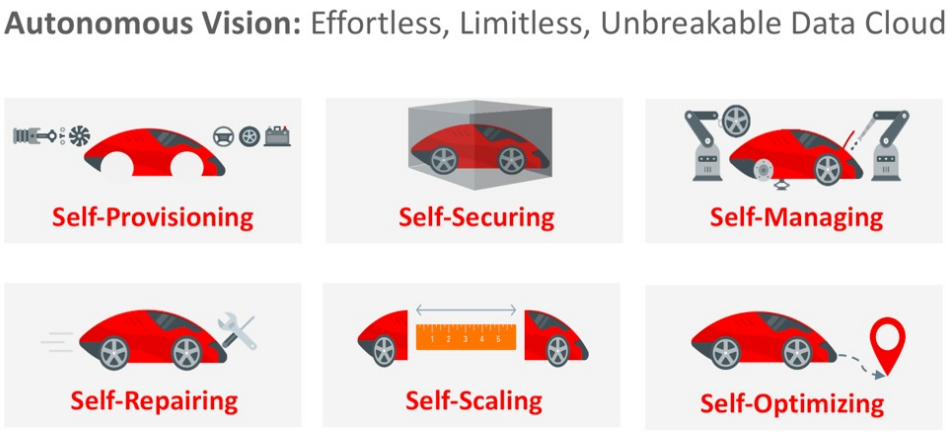
kubernetes.io



Self-healing networks, www.juniper.net



aws.amazon.com/ec2



Autonomous database, www.oracle.com/autonomous-database/what-is-autonomous-database/

No AI/ML Processing

Goals of self-adaptive systems

- A self-adaptive (***self-****) software system is able to self-manage, pursuing the following goals:
- Self-optimize
  - Capability of system to optimize its resource usage or performance while providing its required quality goals
  - E.g., change placement of application components onto system nodes to satisfy application response time
- Self-heal
  - Capability of system to discover, diagnose and recover from faults to provide its required quality goals or degrade gracefully otherwise
  - E.g., detect crashed nodes and exclude them from serving requests

No AI/ML Processing

Goals of self-adaptive systems

- Self-configure
  - Capability of system to automatically integrate new elements, without interrupting the system's normal operation, or tune some configuration parameters
  - E.g., discover new nodes and add them to serve requests

- Self-protect
  - Capability of system to detect anomalies (i.e., intrusion detection) and react to intrusion and attack actions and its consequences (i.e., intrusion response) so to protect from security threats
  - E.g., in a network of IoT devices detect a jamming attack that corrupts network traffic and adapt packets schedule

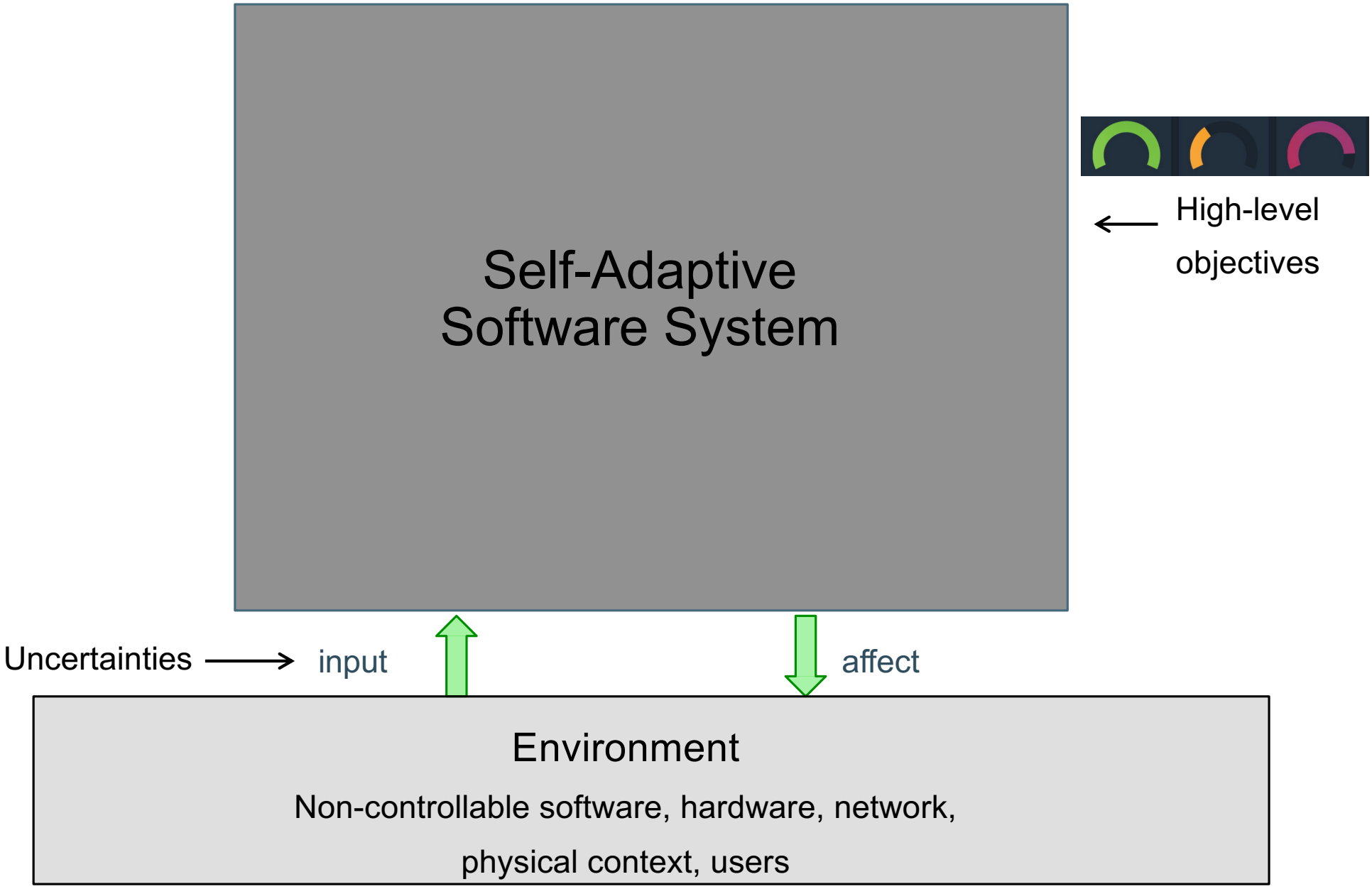
How to achieve the goals of a self-* system?

- The system should know its internal state (self-awareness) and the current external operating conditions (self-situation)
- Should identify changes regarding its state and the surrounding environment (self-monitoring)
- And should adapt consequently (self-adjustment)

- These attributes are the implementation mechanisms

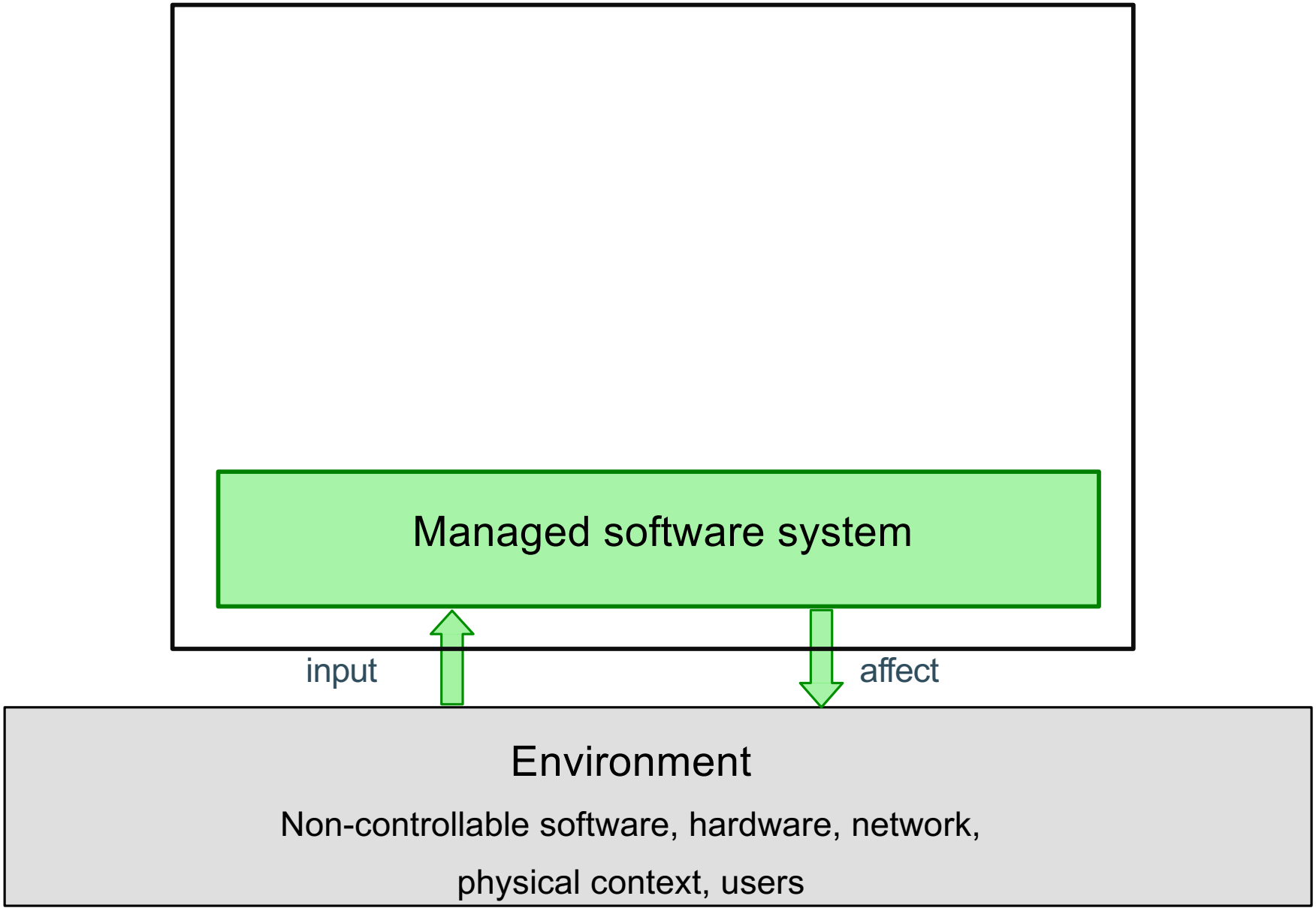
No AI/ML Processing

Conceptual model of self-adaptive system



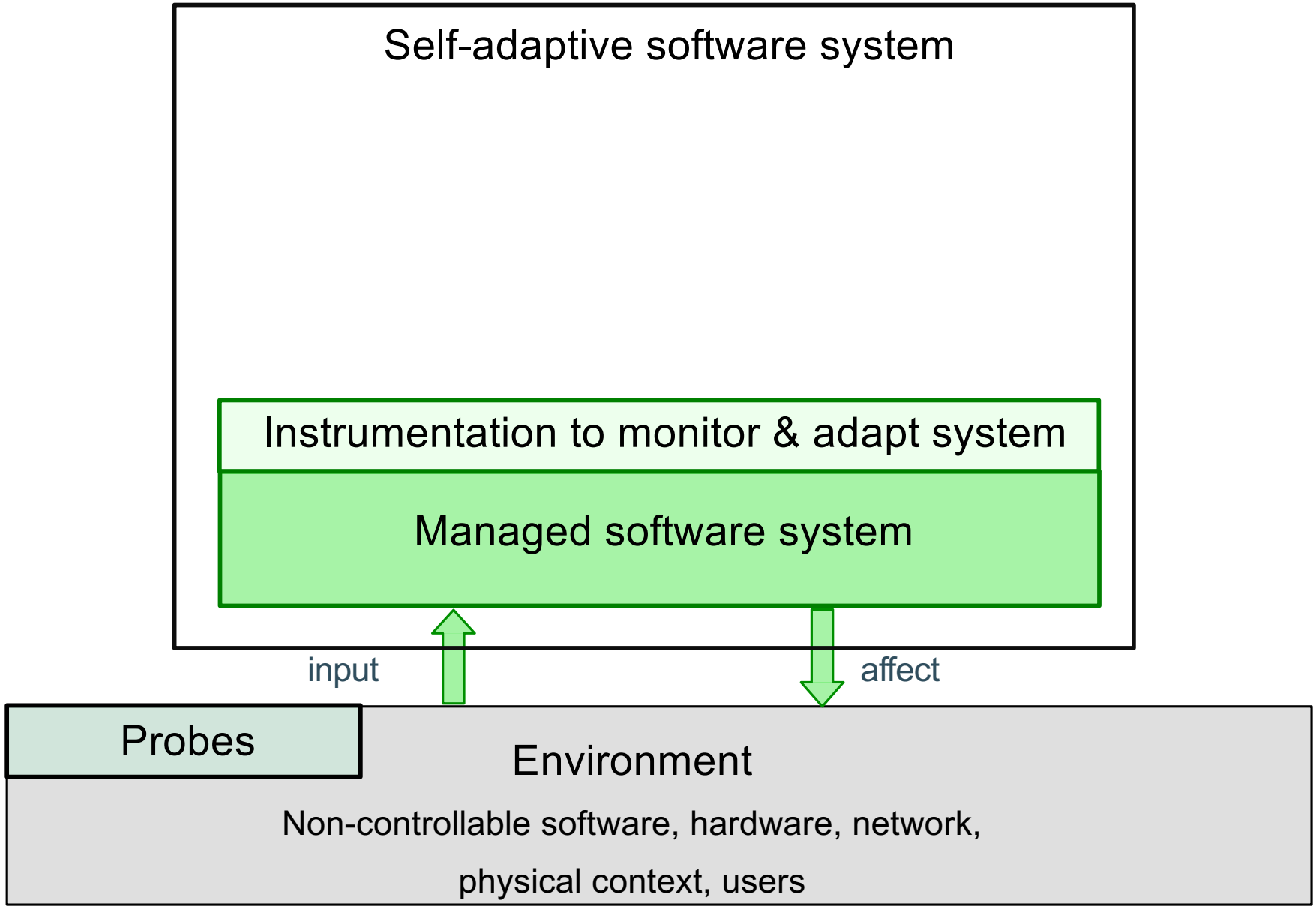
No AI/ML Processing

Conceptual model of self-adaptive system



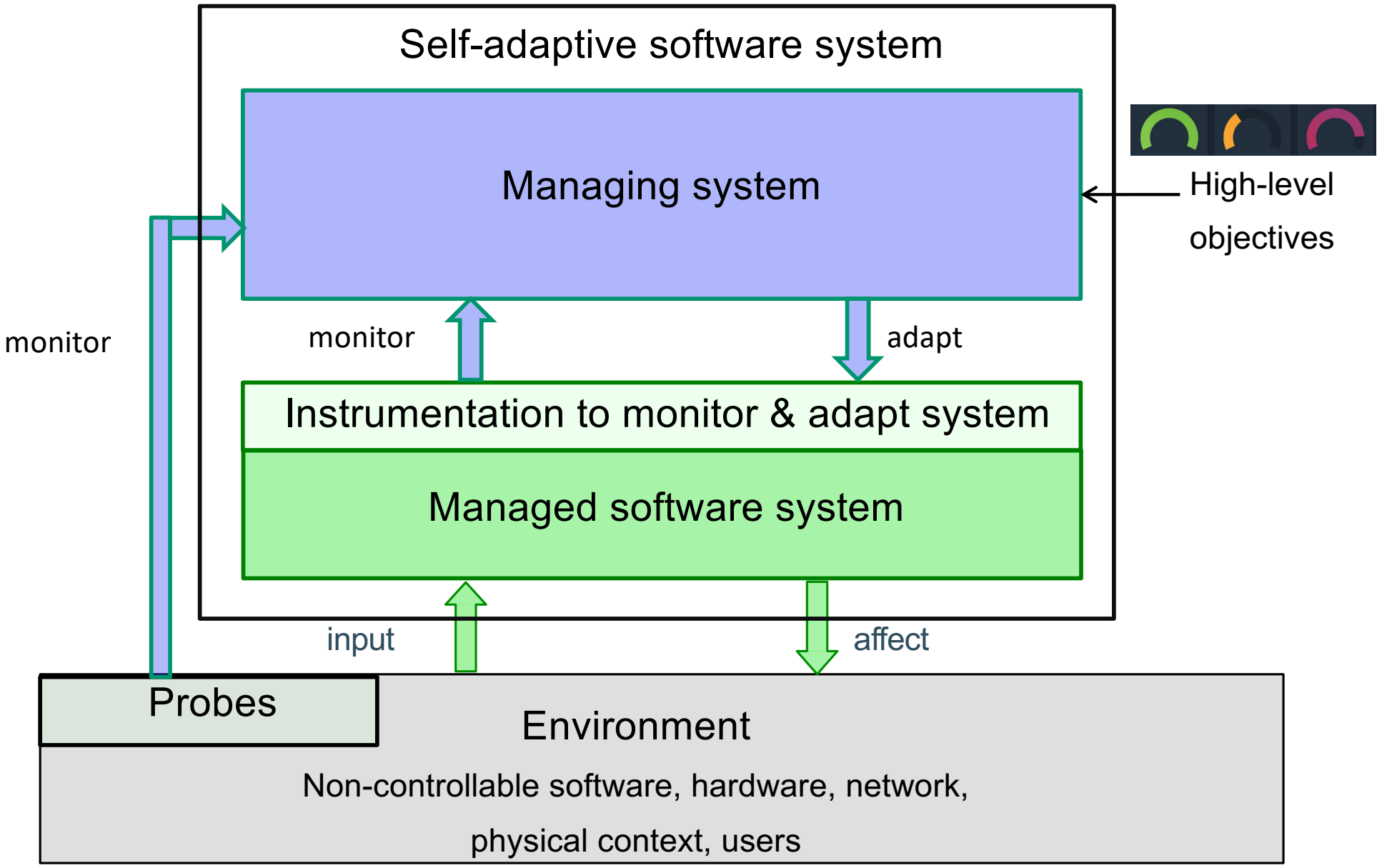
No AI/ML Processing

Conceptual model of self-adaptive system



No AI/ML Processing

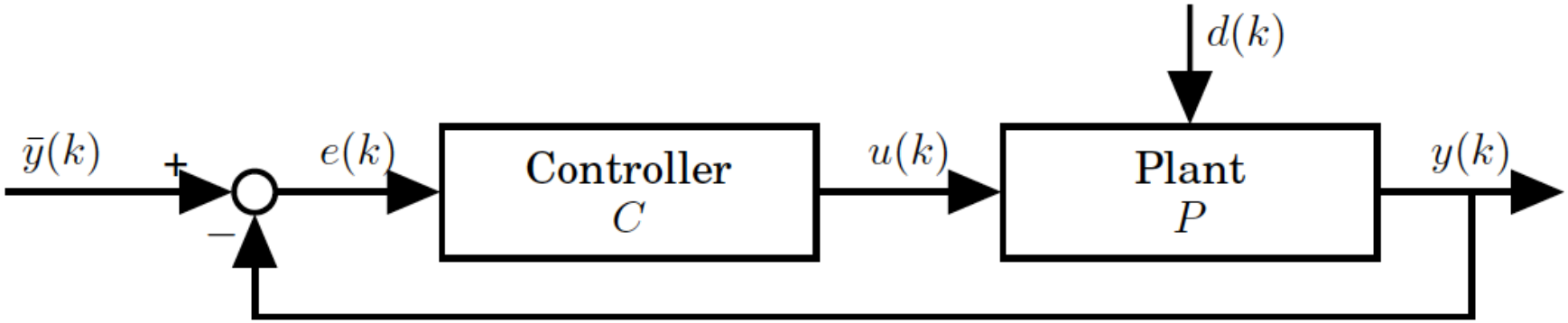
Conceptual model of self-adaptive system



No AI/ML Processing

You are already familiar with this model

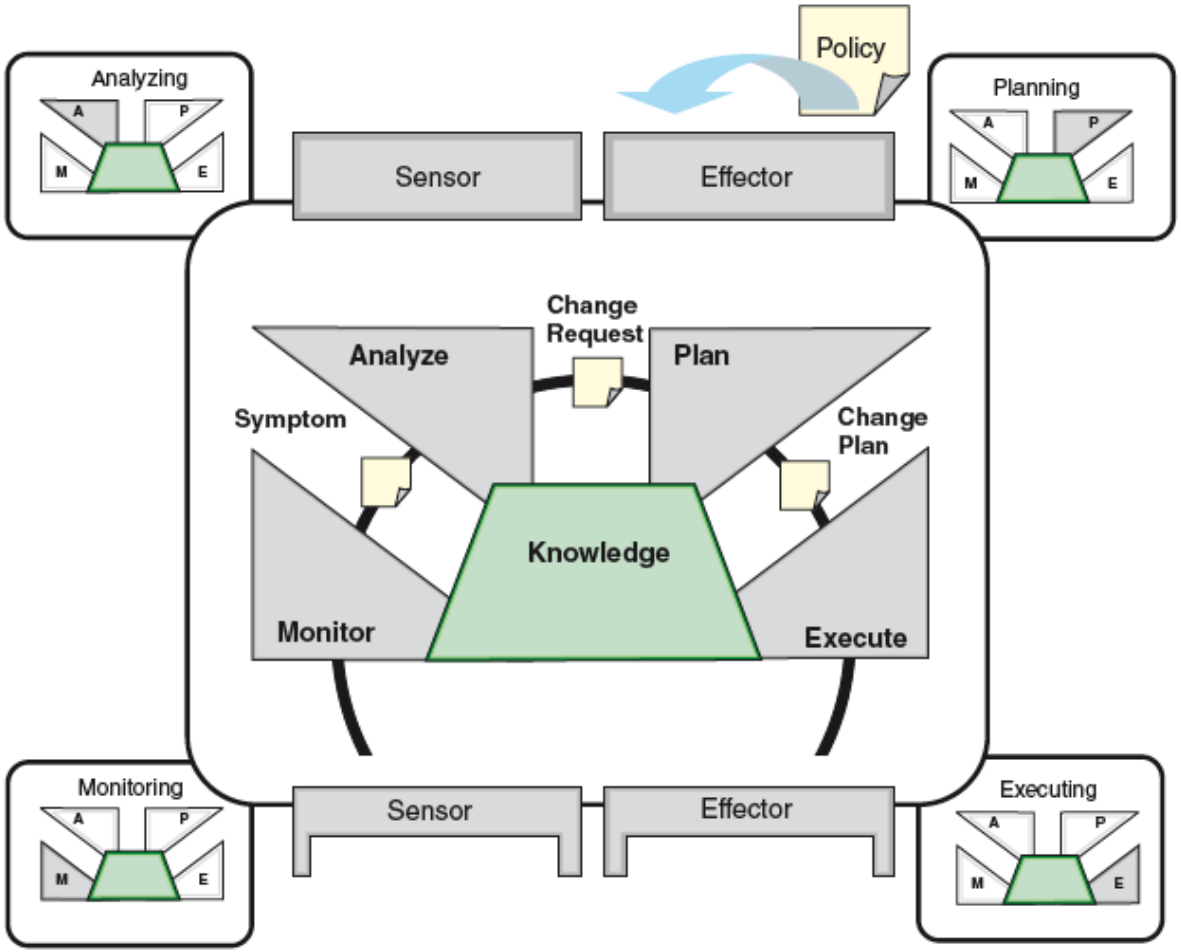
- The control-theory perspective of a self-adaptive system



No AI/ML Processing

MAPE: reference architecture for self-adaptive system

- **MAPE** (Monitor, Analyze, Plan, Execute) loop



No AI/ML Processing

MAPE: building blocks (or phases)

- **M**onitor
  - Collects data from the managed system and execution environment through sensors; aggregates, filters and correlates these data into symptoms that can be analyzed
- **A**nalyze
  - Observes and analyzes situations to determine need for adaptation
  - If adaptation is required, it triggers Plan
- **P**lan
  - Determines which mitigation actions need to be performed so to enact a desired alteration in the managed system
- **E**xecute
  - Enacts the change plan by carrying out the actions determined by Plan through effectors so to adapt the managed system
- Plus **K**nowledge (MAPE-K)
  - Stores shared knowledge regarding relevant aspects of the managed system, environment, and the administrator's goals

No AI/ML Processing

MAPE: Monitor

- Main design options for Monitor:
  - *When* to monitor: continuous (real-time streaming) vs. on demand (event-driven or periodic polling)
  - *What* to monitor: resource utilization, workload dynamics, system performance, failure rates
  - *How* to monitor: architecture (centralized vs. decentralized), methodology (active vs. passive)
  - *Where* to store monitored data (e.g., time-series databases optimized for temporal metrics) and *what* to store (raw data vs. pre-processed data)

No AI/ML Processing

MAPE: Analyze

- Main design options for Analyze:
  - *When* to analyze: event-triggered (e.g., threshold violation) vs. time-triggered
  - Reactive vs. proactive adaptation
    - *Reactive*: triggers adaptation actions *in response to* events that have already occurred (e.g., scaling out resources *after* a workload spike is detected)
    - *Proactive*: relies on *prediction* to plan adaptation actions *in advance* (e.g., scaling out resources *before* an anticipated workload surge occurs)

No AI/ML Processing

MAPE: Plan

- The most challenging and studied phase
- A variety of methodologies can be used to plan adaptation actions
- Optimization theory
  - ✓ Optimal adaptation actions
  - ✗ Problem formulation can be NP-hard, too expensive at runtime
- Heuristics
  - ✓ Faster
  - ✗ Sub-optimal adaptation actions
- Control theory and queueing theory
  - ✓ Provide rigorous mathematical frameworks for modeling system dynamics
  - ✗ Require simplified mathematical abstractions (e.g., system linearity, specific traffic distributions) that struggle with complex system behaviors

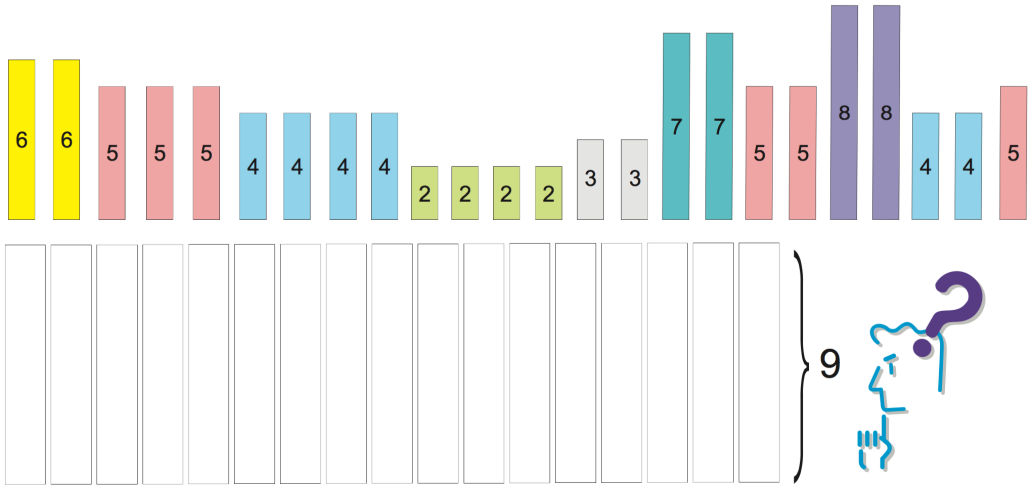
MAPE: Plan

- A variety of methodologies can be used to plan adaptation actions
- Machine learning (including reinforcement learning)
  - RL: learns and adapts policies dynamically from system and environmental interactions
  - ✓ Handles complex non-linear dynamics
  - ✗ Suffers from training overhead, cold-start latency, and black-box behavior
- Large Language Models (LLMs)
  - ✓ Leverages natural language and contextual awareness to interpret system state and generate adaptation plans
  - ✗ Suffer from non-deterministic outputs, high inference latency, and expensive compute footprint

No AI/ML Processing

Plan example: VM/container placement

- Resource management example: optimal bin packing and heuristic policies for the dynamic placement of virtual machines or containers across
- Problem formulation
  - Input: a set of physical servers (bins) with finite capacities and a set of virtual machines or containers (items) with specific resource demands
  - Objective: determine the optimal placement of items across available servers



No AI/ML Processing

Plan example: VM/container placement

- Bin packing: pack items of varying sizes into the minimum number of bins, ensuring that the aggregate resource demand in any given bin does not exceed its capacity https://en.wikipedia.org/wiki/Bin_packing_problem
- Formulated as integer optimization problem, which is NP-hard
- Practical requirement: efficient heuristics to change placement decisions at runtime

$$\text{minimize } K = \sum_{j=1}^{n} y_j$$

$$\text{subject to } K \geq 1,$$

$$\sum_{i \in I} s(i) x_{ij} \leq B y_j, \quad \forall j \in \{1, \ldots, n\}$$

$$\sum_{j=1}^{n} x_{ij} = 1, \qquad \forall i \in I$$

$$y_j \in \{0, 1\}, \qquad \forall j \in \{1, \ldots, n\}$$

$$x_{ij} \in \{0, 1\}, \qquad \forall i \in I \, \forall j \in \{1, \ldots, n\}$$

where $y_j = 1$ if bin j is used and $x_{ij} = 1$ if item i is put into bin j.

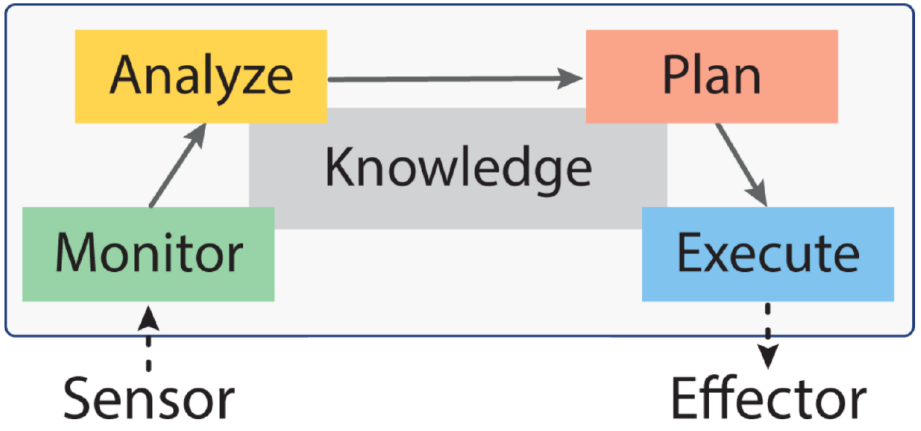
Plan example: VM/container placement

- *Round robin:* organizes bins in a *circular* list and places each item in the next available bin starting from the last used position
  - Fast, but does not minimize the number of used bins
- Bin packing: multiple heuristic policies exist
- *First fit:* scans bins sequentially from the beginning and places each item into the first bin with sufficient residual capacity
  - First fit decreasing: variant where items are sorted in decreasing capacity
- *Best fit*: places the item in the bin with the least residual capacity that still fits the item
  - Best for consolidation
- *Worst fit*: places the item in the bin with the most residual capacity where it fits

No AI/ML Processing

MAPE: architectures

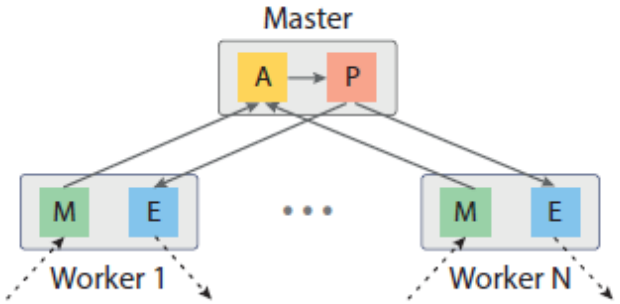
- Architectural design of the managing system
- Centralized MAPE
  - All MAPE components are on a single node
  - Simple, but lacks scalability and fault tolerance, especially in geo-distributed environments
- Decentralized MAPE
  - MAPE components are distributed across multiple nodes
  - Various architectural patterns, each with its own trade-offs
    - No one-size-fits-all solution: the choice depends on system characteristics, application requirements, and operational context



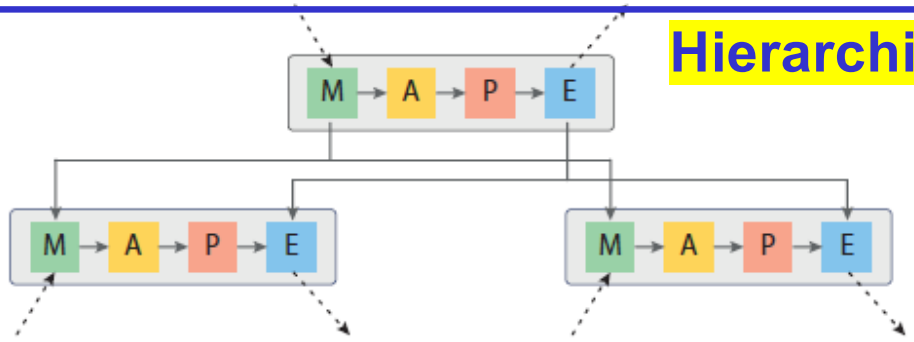
No AI/ML Processing

How to decentralize MAPE

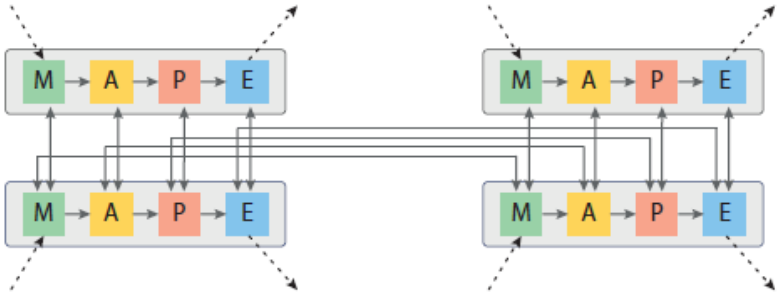
- Architectural patterns for decentralized MAPE



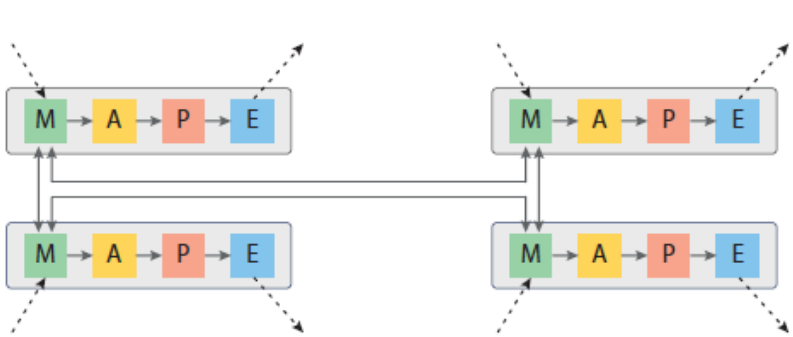
Hierarchical MAPE: master-worker pattern



Hierarchical MAPE: hierarchical control pattern



Flat MAPEs: coordinated control pattern



Flat MAPEs: information sharing pattern

Weyns et al., On patterns for decentralized control in self-adaptive systems. In *SEfSAS II*, 2013

No AI/ML Processing

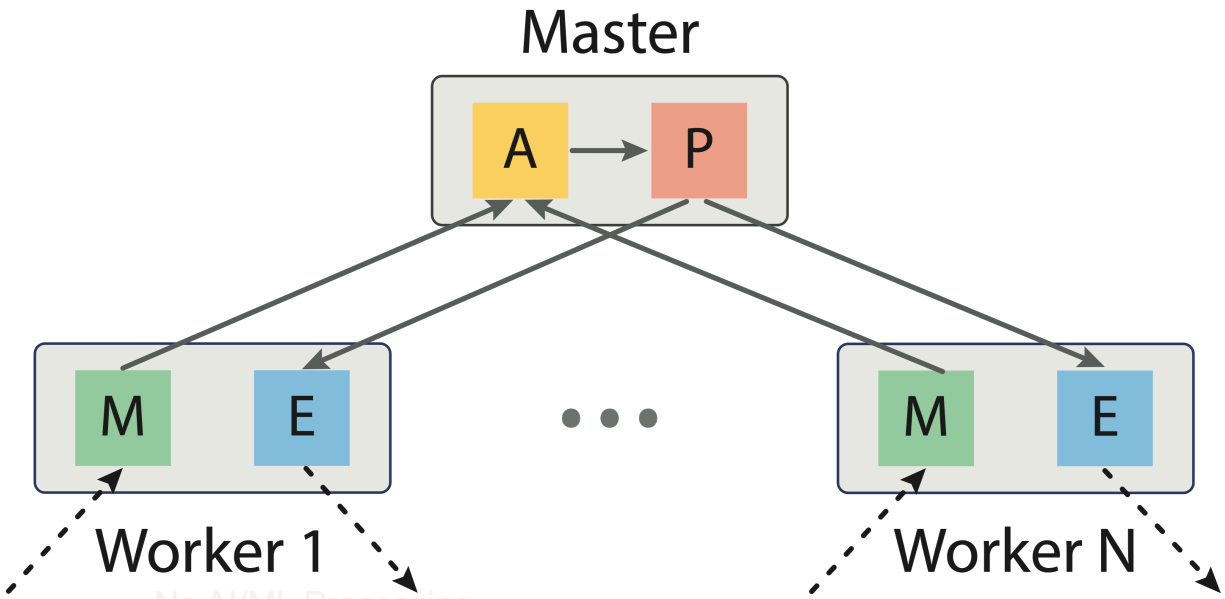
Decentralized MAPE patterns

- Design choice: ***hierarchical*** vs. ***flat***
- Hierarchical MAPE patterns: multiple MAPE loops organized in tiers, where higher-level loops manage subordinated loops
  - Master-worker
  - Hierarchical control
- Flat MAPE patterns: multiple MAPE loops cooperate as peers
  - Coordinated control
  - Information sharing
- Pros and cons
  - Hierarchical: easier to design and coordinate, but risk of bottleneck in top level of hierarchy
  - Flat: more scalable and resilient, but more complex to coordinate

No AI/ML Processing

Hierarchical MAPE: master-worker pattern

- Distribute M and E on workers, keep A and P centralized on master
- ✓ Global view on master who can achieve global adaptation goals
- X Communication overhead and risk of performance bottleneck and SPOF on master
  - Can scale across multiple layers, with upper master layers



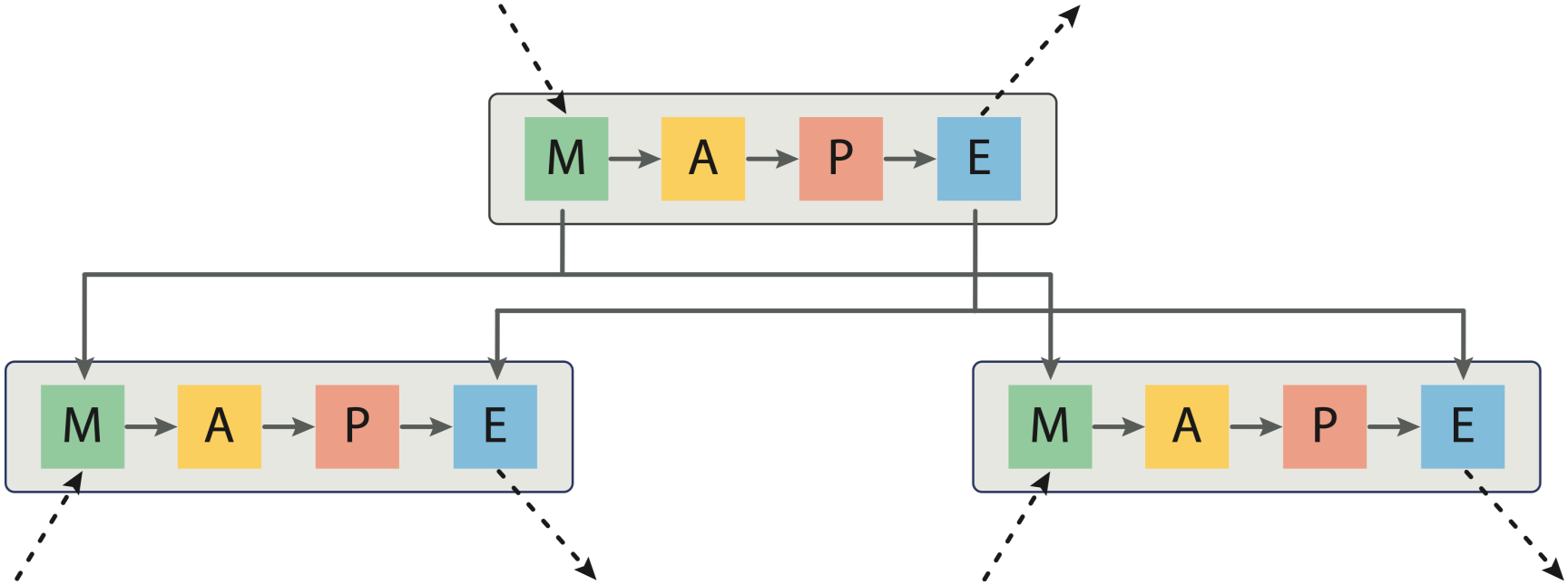
No AI/ML Processing

Hierarchical MAPE: hierarchical control pattern

- Multiple MAPE loops, which can operate at different time scales and with separation of concerns

✓ Top-level MAPE can achieve global goals, increased flexibility

✗ Can be non-trivial to identify different levels of control, depends on managed system characteristics



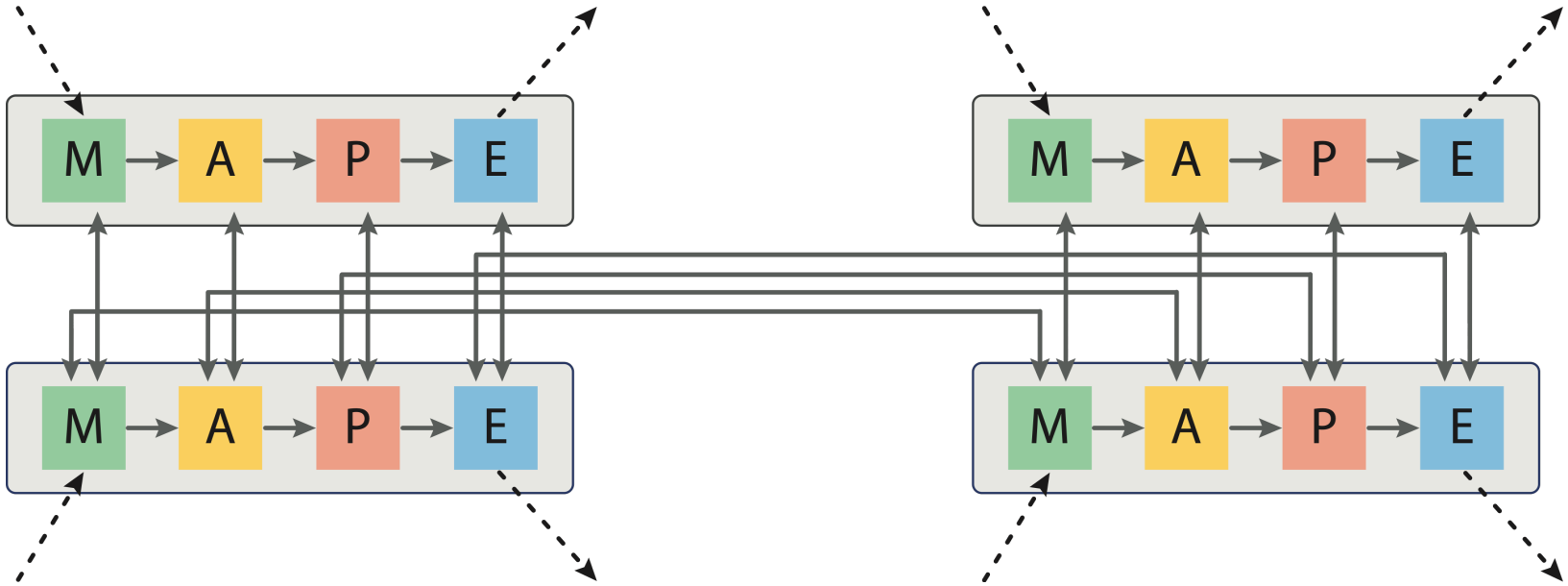
No AI/ML Processing

Flat MAPE: coordinated control pattern

- Multiple control loops, each one in charge of some part of the managed system but coordinated through interaction

✓ Better scalability

✗ More complex to take joint adaptation decisions



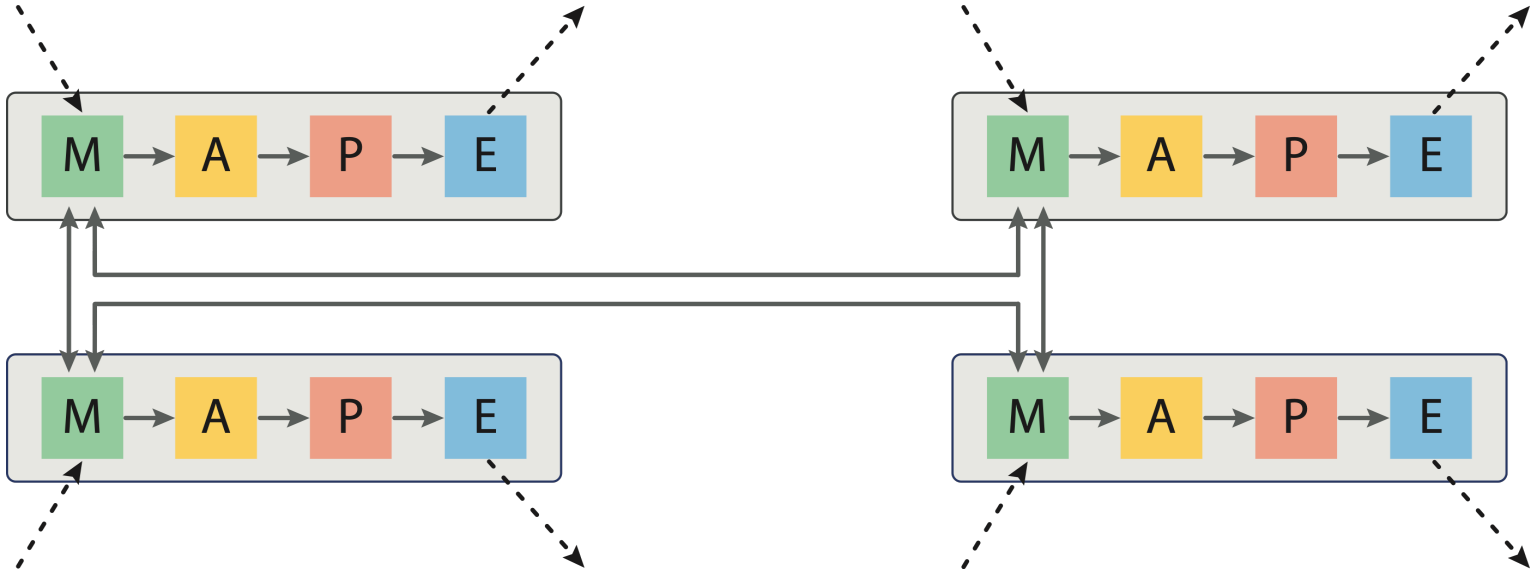
No AI/ML Processing

Flat MAPE: information sharing pattern

- Special case of coordinated control pattern: interaction only among M components

✓ Better scalability

✗ Lack of coordination on planning, conflicting or sub-optimal adaptation actions can be enacted



No AI/ML Processing

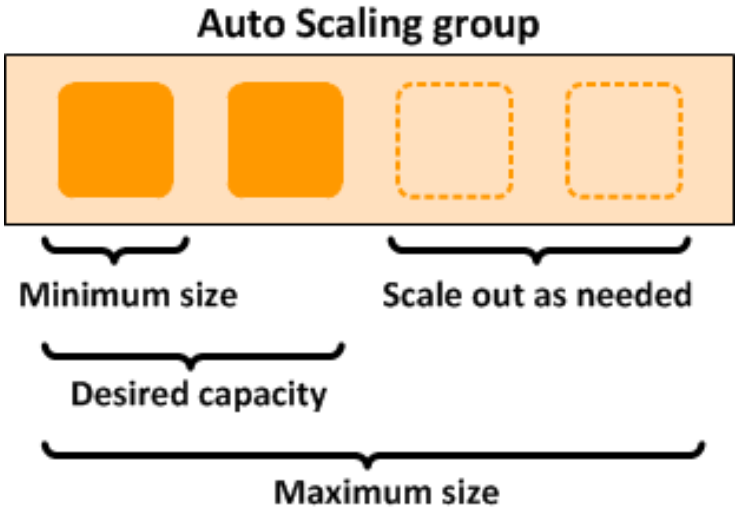
Case study: self-adaptive resource management

- We analyze 3 examples of self-adaptive systems for resource management
  1. Auto-scaling AWS EC2 instances
  2. Self-adaptive selection of application component services
  3. Auto-scaling microservice-based applications
- Common ground
  - Systems operate under uncertainty (e.g., workload spikes, node crashes)
  - Adaptation goal: maintain target SLOs (e.g., application response time, availability, cost)
- The examples span varying planning methodologies and MAPE architectures

No AI/ML Processing

Example 1: AWS EC2 Auto Scaling

- AWS service that automatically adds or removes EC2 instances based on user-defined conditions and health checks https://aws.amazon.com/ec2/autoscaling/



- **M**APE Monitor: monitor scaling metrics via AWS CloudWatch https://aws.amazon.com/cloudwatch/
- Scaling metrics can be
  - Standard: CPU utilization, memory usage, network throughput, Application Load Balancer (ALB) request count
  - Application-specific custom metrics, e.g., queue depth, request rates, active sessions

Example 1: AWS EC2 Auto Scaling

- MA**P**E Plan: we consider multiple policies provided by AWS Auto Scaling
  https://docs.aws.amazon.com/autoscaling/ec2/userguide/scale-your-group.html
  - Threshold-based policy: heuristic policy with scaling actions triggered by thresholds defined on scaling metric
  - Target-based policy: heuristic policy with scaling actions based on target metric value
  - Predictive policy: ML-based policy which forecasts traffic and proactively adjusts capacity
- Policy latency depends on VM provisioning overhead
  - Even minutes, depending on VM type
- Not only auto-scaling, but also self-healing
  - AWS Auto Scaling continuously monitors EC2 instances, detects unhealthy ones and terminates them, and provisions replacements to maintain auto-scaling group size

Example 1: Threshold-based policy

- Threshold-based policy: rule-based policy based on predefined thresholds
- A *reactive* heuristic
  - Specify upper and lower thresholds on some scaling metric(s)
  - Upper threshold → triggers scale-out
  - Lower threshold → triggers scale-in

```
if (metric > upper_thr) scale-out
else if (metric < lower_thr) scale-in
```

    - Example of *scale-out* rule: if average CPU utilization of all instances exceeds 70% for 1 minute, add 1 instance
    - Example of *scale-in* rule: if average CPU utilization of all instances is below 35% for 5 min, remove 1 instance
  - CloudWatch monitors and sends alarms when scaling metric(s) crosses upper or lower thresholds during the specified evaluation period

No AI/ML Processing

Example 1: Threshold-based policy

- AWS Auto Scaling provides 2 threshold-based policy variants
- Simple scaling
  - Applies a fixed capacity adjustment (e.g., +1, -1) when a CloudWatch alarm triggers, followed by a cooldown period
    - Cooldown period: delay after a scaling action to allow metrics to stabilize before subsequent actions are evaluated
- Step scaling
  - Applies different capacity adjustments depending on how far the metric exceeds a threshold
  - ✓ More responsive scaling

No AI/ML Processing

Example 1: Threshold-based policy

✓ Simple and easy-to-understand
  - Select scaling metric(s), thresholds, and evaluation period

✗ Metric selection requires application knowledge
  - Metrics are application-dependent: CPU-, memory-, I/O-intensive workload, or even a mix

✗ No application-specific metrics (e.g., response time)

✗ Thresholds require experience for manual tuning
  - Thresholds can be too aggressive or conservative (next slide)
  - Performance depends heavily on their setting

✗ May be not robust to dynamic workloads

💡 Idea: exploit reinforcement learning to learn optimal threshold values

Dynamic multi-metric thresholds for scaling applications using reinforcement learning, *IEEE TCC 2023* https://doi.org/10.1109/TCC.2022.3163357

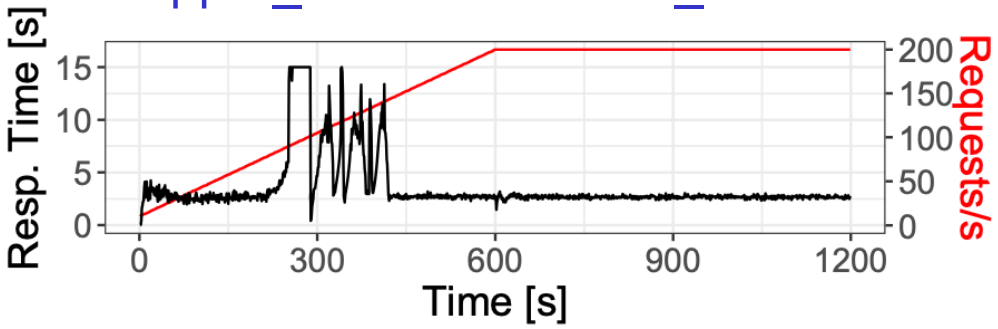
No AI/ML Processing

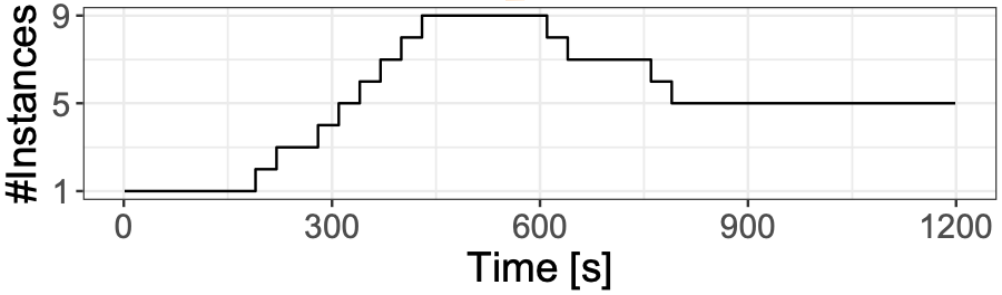
Example 1: Threshold-based policy

- Examples of wrong threshold setting
  - Scaling metric is CPU utilization

evaluation period = 30 s.
upper_thr = 80% lower_thr = 25%

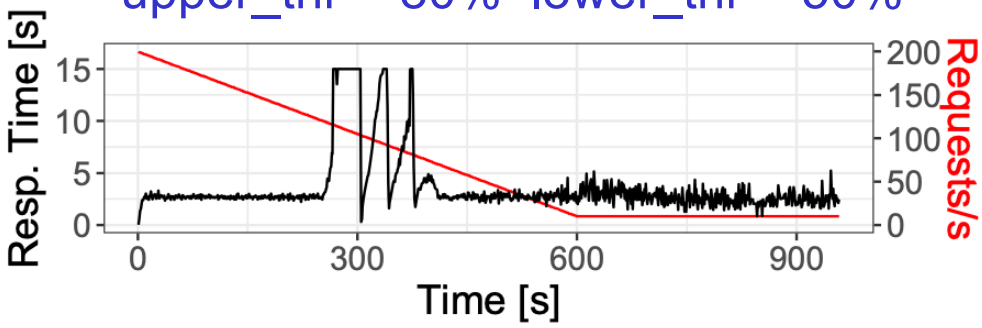
over-provisioned

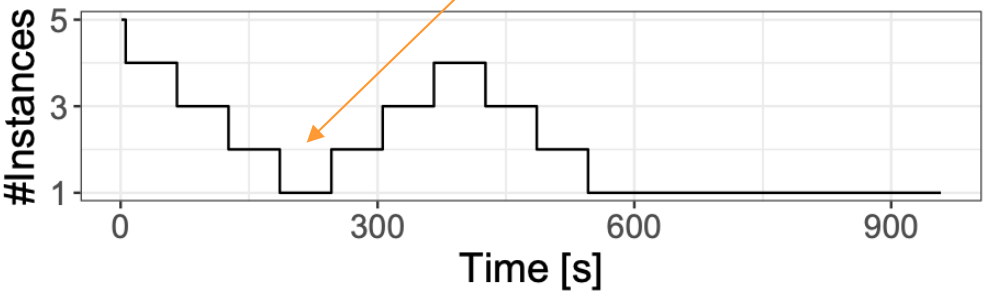




evaluation period = 60 s.
upper_thr = 80% lower_thr = 50%

under-provisioned





Autoscaler Evaluation and Configuration: A Practitioner's Guideline, *ICPE 2023*
https://research.spec.org/icpe_proceedings/2023/proceedings/p31.pdf

No AI/ML Processing

Example 1: Target-based policy

- Target tracking scaling: heuristic policy based on target
- A *reactive* heuristic
  - Specify a target value for a metric, and AWS Auto Scaling automatically adjusts capacity to keep the metric close to target
  - Example: maintain average CPU utilization at 50%

✓ Simpler configuration: specify the desired target rather than multiple thresholds

✓ Industry relevance

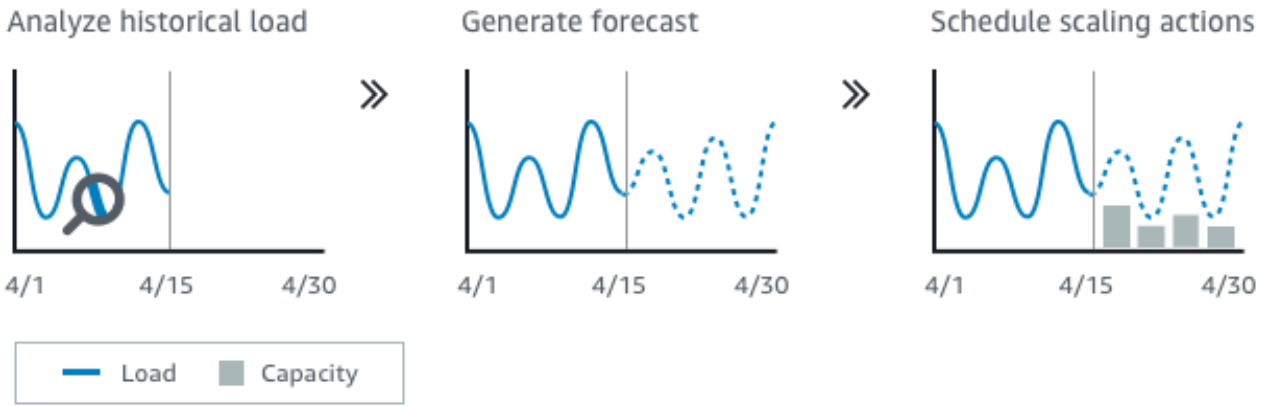
  - Kubernetes Horizontal Pod Autoscaler (HPA) employs an analogous reactive policy for pod replica scaling
  - HPA scales number of pods according to ratio between observed value and target value

$$desiredReplicas = \left\lceil currentReplicas \frac{currentMetricValue}{desiredMetricValue} \right\rceil$$

No AI/ML Processing

Example 1: Predictive policy

- A *proactive* ML-based policy
  - ML model trained to forecast workload or resource demand, capturing daily and weekly patterns
    - Scaling metrics: ALB request count, CPU utilization
  - Requires historical data collected via CloudWatch
    - Minimum of 1 day of data to initiate predictions
    - Hourly forecasts for the next 48 hours based on the past 14 days
    - Forecast updated every 6 hours
  - Scaling action: scale out to reach a target utilization



No AI/ML Processing

Example 1: Predictive policy

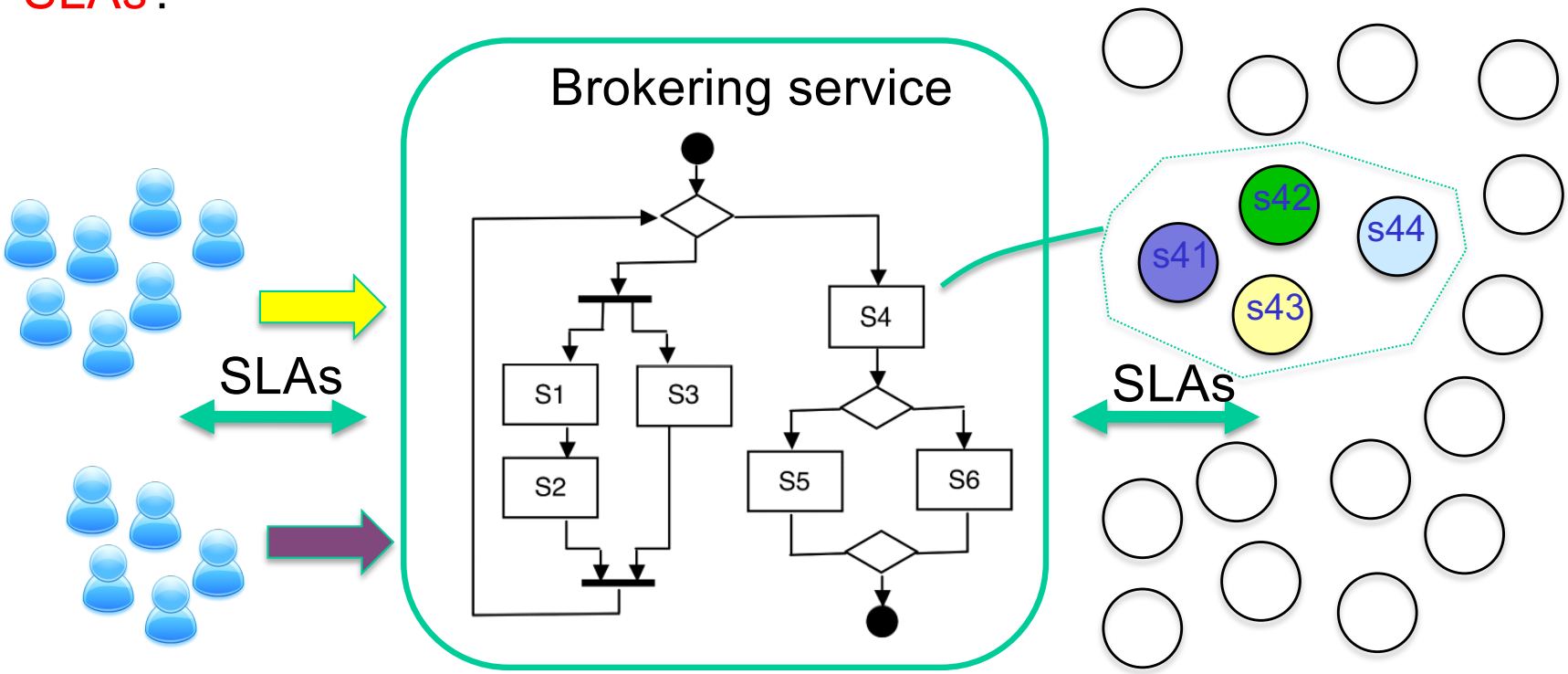
- ✓ Proactive
  - Scales compute capacity in advance rather than reacting after performance deteriorates
- ✓ Dependent on ML training data
  - Larger volumes of historical data → better forecast accuracy
- ✗ Effectiveness depends on scaling metric
- ✓ Supports only scale-out decisions
  - Separate policy to scale-in

- All these policies assume that VM instances are homogeneous

No AI/ML Processing

Example 2: Self-adaptive service selection

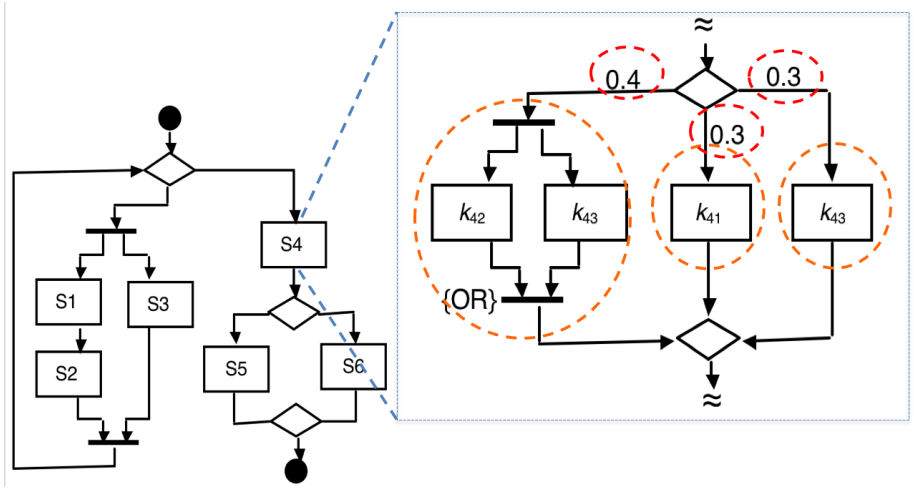
- QoS-driven self-adaptation of SOA applications
  - Each abstract service may have multiple concrete implementations with different non-functional properties
  - Which concrete services should be selected to meet application SLAs?

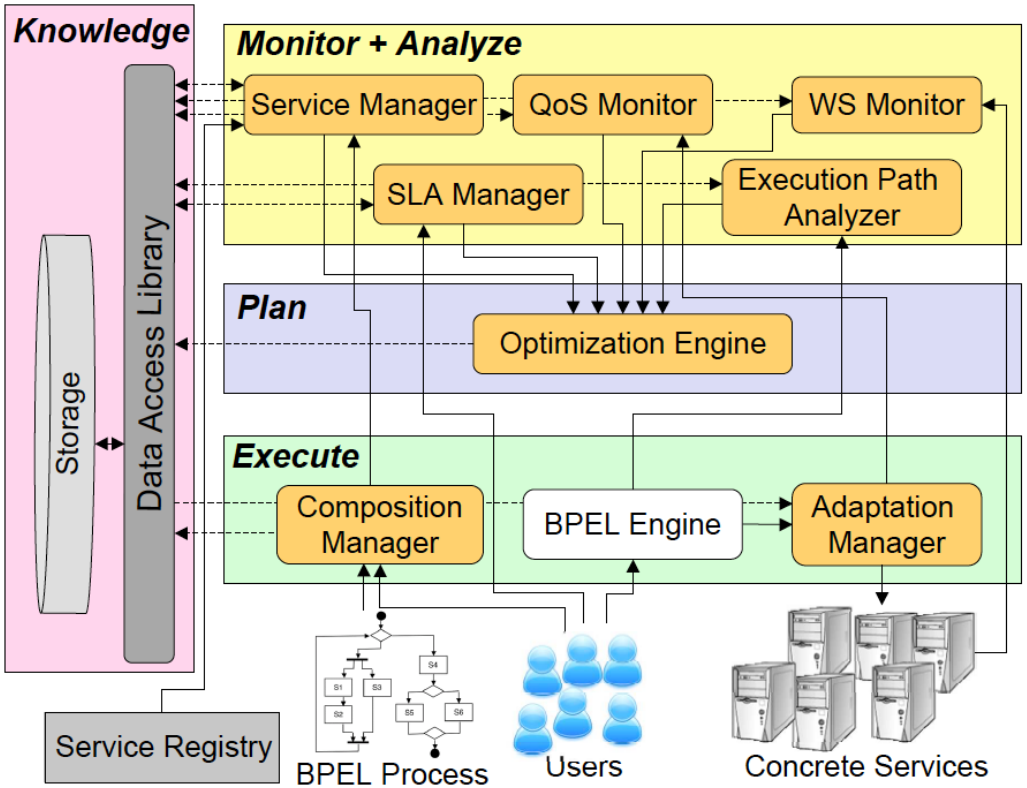


MOSES: a framework for QoS driven runtime adaptation of service-oriented systems, *IEEE TSE* 2012 http://www.ce.uniroma2.it/publications/tse2012.pdf

Example 2: Self-adaptive service selection

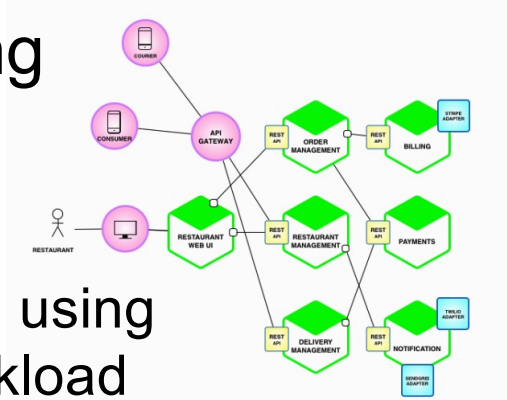
- MA**P**E Plan
  - Centralized policy: find the *optimal set of concrete services* (and their *coordination*) using linear programming optimization

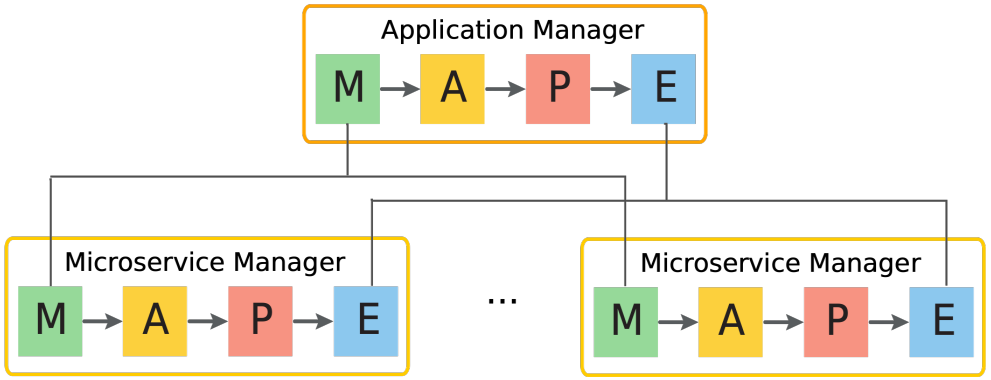


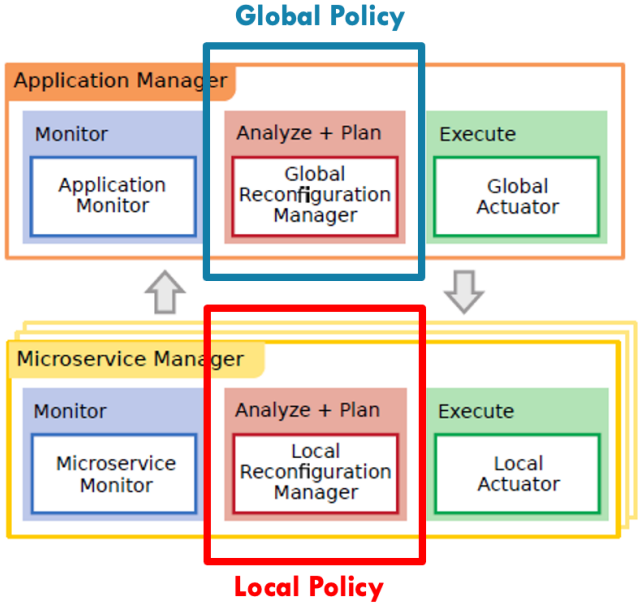


Example 3: Hierarchical auto-scaling of microservices

- Hierarchical control pattern for elastic scaling of microservices-based application
  - Goal: app response time < R_{max}
  - Each Microservice Mngr. determines scale-out/in using a local policy based on queueing theory and workload prediction and sends scaling proposal to Application Mngr.
  - Application Mngr. coordinates proposals by approving or rejecting them and sends decisions to local Execute components







Hierarchical scaling of microservices in Kubernetes, *ACSOS 2020*
http://www.ce.uniroma2.it/publications/acsos2020.pdf

References

- The vision of autonomic computing
  https://ieeexplore.ieee.org/document/1160055
- An introduction to self-adaptive systems: A contemporary software engineering perspective, chapter 1
  https://introsas.cs.kuleuven.be/2020ExcerptBook.pdf
- On patterns for decentralized control in self-adaptive systems
  https://www.ics.uci.edu/~seal/publications/2012aSefSAS.pdf

No AI/ML Processing