# Mathematical Optimization of Hybrid Software Architectures

Introduction for New Team Members

INPT - CEDoc 2TI - SEEDS Research Team

Supervisor: Prof. Driss ALLAKI

*"A good decomposition is not just clean code*

*-- it is a mathematical optimum."*

# 1. What is Software Architecture?

Software architecture is the high-level structure of a software system. It defines how the system is divided into components, how those components interact, and what principles govern their design and evolution.

Think of it like the floor plan of a building: before laying any bricks (writing code), you need a blueprint that defines rooms (components), hallways (interfaces), and how everything connects (dependencies).

## Why Does Architecture Matter?

- Maintainability: A well-structured system is easier to understand and modify.
- Scalability: Good architecture allows parts of the system to scale independently.
- Team Organization: Architecture defines team boundaries -- each team owns a component.
- Reliability: Loosely coupled components mean a failure in one part does not crash everything.

## The Two Main Paradigms

**Monolithic Architecture:** All functionality lives in a single codebase and is deployed as one unit. Simple to develop initially, but becomes hard to manage as the system grows. Example: a traditional Java EE application deployed as a single WAR file.

**Microservice Architecture:** The system is broken into small, independent services that communicate over a network (typically HTTP/REST or gRPC). Each service can be developed, deployed, and scaled independently. Example: Netflix, Amazon, Uber.

## 2. Kubernetes, Containers & Deployment

Modern software deployment relies heavily on containerization and orchestration:

**Containers (Docker):** A lightweight, portable way to package an application with all its dependencies. Think of it as a standardized shipping container for software -- it works the same everywhere.

**Kubernetes (K8s):** An orchestration platform that manages containers at scale. It handles deployment, scaling, load balancing, and self-healing. It is the de facto standard for running microservices in production.

**Pod:** The smallest deployable unit in Kubernetes. Contains one or more containers that share storage and network. A microservice typically runs in a Pod.

**Service:** A Kubernetes abstraction that provides a stable network endpoint for a set of Pods. Enables service-to-service communication.

The key insight: Kubernetes makes microservice deployment practical, but it does NOT tell you HOW to decompose your system into microservices. That decomposition decision is exactly what this project addresses.

# 3. The Coupling-Cohesion Problem

The fundamental challenge in software architecture is finding the right decomposition -- how to split a system into components.

## Key Concepts

**Coupling:** The degree of interdependence between two modules. High coupling means modules are tightly connected and changes in one ripple through to others. We want LOW coupling between modules.

**Cohesion:** The degree to which elements within a module belong together. High cohesion means a module has a single, well-defined purpose. We want HIGH cohesion within modules.

**Modularity:** A quantitative measure that captures the quality of a partition. High modularity means dense connections within clusters and sparse connections between them. Formally: Q = (1/2m) * Sum[A_ij - (k_i * k_j / 2m)] * delta(c_i, c_j)

## The Trade-Off

If you make every component its own microservice: coupling is minimized but you have massive overhead from network communication, distributed transactions, and operational complexity.

If you keep everything in one monolith: cohesion is trivially maximized but you lose scalability and team independence.

The OPTIMAL solution is somewhere in between -- a hybrid architecture where some components are grouped into monolithic modules and others are separated as microservices. Finding this optimum is a combinatorial optimization problem.

# 4. Existing Approaches & Their Limits

## Manual Decomposition

Most teams decompose systems based on developer intuition, domain knowledge, or organizational boundaries (Conway Law). This works for small systems but fails at scale -- there are too many possible decompositions to evaluate manually.

## Domain-Driven Design (DDD)

Eric Evans DDD proposes decomposing by "bounded contexts" -- business domains. This gives good semantic boundaries but provides no mathematical guarantee of optimality. Two DDD experts may disagree on the correct decomposition.

## Static Analysis Tools

Tools like SonarQube, Structure101, or JDepend analyze code dependencies and flag high coupling. They detect PROBLEMS but do not suggest SOLUTIONS.

## What is Missing?

All existing approaches lack a mathematical framework that:

- Formally defines what "good decomposition" means (objective function)
- Explores the entire solution space systematically
- Provides provably optimal or near-optimal solutions
- Allows quantitative comparison of different decompositions

# 5. Our Mathematical Approach

This project models the decomposition problem as a GRAPH PARTITIONING problem and applies three distinct optimization strategies:

## The Model

- Each software component is a NODE in a graph
- Dependencies between components are weighted EDGES
- Edge weight = strength of coupling (0 to 1)
- The GOAL: partition nodes into clusters that maximize intra-cluster cohesion and minimize inter-cluster coupling

## Three Solution Strategies

**1. Spectral Clustering:** Uses the eigenvalues and eigenvectors of the graph Laplacian matrix. The "eigengap heuristic" automatically suggests the optimal number of clusters. Rooted in linear algebra and spectral graph theory.

**2. Genetic Algorithm:** An evolutionary approach where candidate partitions evolve through selection, crossover, and mutation. Can optimize ANY fitness function, not just modularity.

**3. Louvain Algorithm:** A greedy, hierarchical community detection method. Extremely fast (near-linear time) and widely used in network science. Directly maximizes modularity.

## Why Three Methods?

Each method has different strengths: Spectral is mathematically principled, GA is flexible, Louvain is fast. Comparing them gives architects a toolkit for different scenarios.

# 6. Getting Started

## Prerequisites

- Python 3.8+ with pip
- Basic linear algebra (matrices, eigenvalues)
- Basic graph theory (nodes, edges, adjacency)
- Familiarity with optimization concepts

## Setup

Install dependencies:

```
pip install networkx numpy scipy pandas scikit-learn
        matplotlib seaborn pyvis plotly python-louvain
```

## Workflow

- Step 1: Study the synthetic dataset (data/synthetic_dependency_graph.csv)
- Step 2: Run Notebook 1 -- Spectral Clustering
- Step 3: Run Notebook 2 -- Genetic Algorithm
- Step 4: Run Notebook 3 -- Louvain + Final Comparison
- Step 5: Write the technical report

## Key Files

| | |
|---|---|
| **data/:** | Contains the synthetic dependency graph and node-to-cluster mapping |
| **notebooks/:** | Three self-contained Jupyter notebooks -- one per method |
| **reports/:** | Generated figures, CSV results, and your final report |

**Good luck! Remember: read this document fully before starting code.**