2025 IEEE 22nd International Conference on Software Architecture (ICSA) | 979-8-3315-2090-8/25/$31.00 ©2025 IEEE | DOI: 10.1109/ICSA65012.2025.00016

Architecture and Performance Anti-patterns Correlation in Microservice Architectures

Alberto Avritzer
eSulabSolutions Inc.
Princeton (NJ), USA
beto@esulabsolutions.com

Andrea Janes
Free University of Bozen-Bolzano
Bolzano, Italy
ajanes@unibz.it

Catia Trubiani
Gran Sasso Science Institute
L'Aquila, Italy
catia.trubiani@gssi.it

Helena Rodrigues
Universidade do Minho
Braga, Portugal
helena@dsi.uminho.pt

Yuanfang Cai
Drexel University
Philadelphia, USA
yfcai@cs.drexel.edu

Daniel Sadoc Menasché
Federal University of Rio de Janeiro (UFRJ)
Rio de Janeiro, Brazil
sadoc@dcc.ufrj.br

Álvaro José Abreu de Oliveira
Universidade do Minho
Braga, Portugal
a88001@alunos.uminho.pt

***Abstract*—Microservice architecture design requires the architect to meet the needs of multiple stakeholders and to address their needs for maintainability, scalability, and availability. In the microservice architecture context, a comprehensive performance and scalability assessment is a dynamic activity, which is focused on the detection of service level metric deviations from objectives using a defined operational profile. Root cause analysis is focused on the identification of the activated microservice components given the defined load profile. Therefore, performance issues are identified by detecting dynamic deviations from the expected behaviors of the service level metric.**

In contrast, microservice architecture assessment focus is on identifying implicit relations among microservice components. Architecture anti-patterns are identified by detecting deviations from the defined formal design patterns. As the ultimate objective of microservice architecture design is to build high-quality applications it would be expected that architecture refactoring based on the removal of architecture anti-patterns will result in meeting stakeholder needs of better scalability and availability.

In this paper we present an empirical assessment of architecture anti-pattern detection in combination with the identification of performance issues using two state of the art tools: DV8 for architecture and PPTAM for performance. We make use of Train Ticket, i.e., a benchmark microservice system, and we observed the co-occurrence of architectural (Clique) and performance (Blob) anti-patterns, noting that high coupling shows much worse performance scores. We have found strong correlation between the normalized distance performance metric and architecture coupling values using several similarity metrics. Our empirical results show that operational profile based performance testing and analysis can be used to help prioritize architecture refactoring.

***Index Terms*—Microservice architecture, Performance evaluation, Antipattern detection**

I. Introduction

Microservice architecture design structures an application as a collection of small autonomous services with the objective of enhancing development practices and maintainability. It is a complex distributed activity that evaluates *architecture alternatives* for data management, data consistency, communication style, service orchestration, etc. In addition, **horizontal** and **vertical** microservice scaling strategies are used to react to changes in workload. These microservice architecture design choices impact the application's performance and scalability.

Scalability can be characterized as architectural structure scalability or system load scalability. A system architecture is structurally scalable if its architecture design does not restrict the number of objects it can hold simultaneously (e.g., connections, transactions, or users) [1]. In contrast, a system is considered load scalable if it is able to satisfy performance requirements when the offered load is increased [2]. In addition, elastic scaling is proposed to increase the architecture structure in response to an increase in load [3].

A performance and scalability end-to-end assessment approach of microservice architectures entails the application of expertise in the following domains: (1) the definition of the service-level metric objective, (2) the high-level model definition of the system being evaluated, (3) the approach used for automated load generation, (4) the operational profile definition of load types and their probability of occurrence. These are black-box customer focused activities that are complemented by two architecture focused activities: (5) the evaluation of microservice architecture alternatives, and (6) the identification of the architecture microservice resources activated in the performance and scalability assessment. This scalability assessment framework is illustrated in Figure 1.

The architecture and performance assessment of a microservice application can trigger architecture and performance anti-patterns. However, since architecture design rules are mostly structural and static, and performance assessment rules are mostly dynamic, in this paper we address the following research question: "Is there a correlation between architectural and performance issues?"

By answering this research question, we intend to identify which aspects need further research and whether the results already achieved provide a consistent answer on the impact the microservice architecture has on system performance and scalability.

The key contribution of this paper is the **empirical assessment** of the correlation between architecture and performance issues.

The paper is structured as follows. Section II presents an overview of the related work. In Section III, we provide a detailed definition of the performance and scalability analysis framework for the assessment of microservice architectures. Section IV presents the list of software performance anti-patterns we selected to describe the identified performance and scalability bottlenecks. Section V

2835-7043/25/$31.00 ©2025 IEEE
DOI 10.1109/ICSA65012.2025.00016

describes the architecture anti-patterns detection framework, while Section VI describes the performance and scalability anti-pattern detection framework. Section VII presents the empirical evaluation of the scalability assessment framework. Section VIII presents the identified threats to validity. Finally, Section IX contains our conclusions and guidance for future research.

II. Related Work

A. Software Performance Anti-patterns

Software performance anti-patterns detection has been investigated looking at different software systems' abstractions, such as the architectural models, the source code analysis, and the data analysis of performance testing results.

1) Architectural models: In [4] a first-order logic formalization of software performance anti-patterns is provided, and its associate engine is used to detect anti-patterns in UML models enriched with MARTE profile for performance-related knowledge. In [5] an approach for the automated detection and solution of software performance anti-patterns in Palladio architectural models was presented. The experimentation showed an observed performance improvement of up to 50% in a case study, thus illustrating the effectiveness of performance anti-pattern solutions. In [6] a model-driven approach to detect performance anti-patterns in Architecture Description Languages (ADL) is presented, and experimental results give evidence of the anti-patterns' benefits in ADL-based software architectures. In [7] software performance anti-patterns are adopted to build traceability links between architectural models and performance analysis results, thus supporting software architects in the identification of the most critical (from a performance-based perspective) design elements. In [8] the authors introduced a tool to automatically detect software performance anti-patterns in UML models.

2) Source code analysis: In [9] a framework for the automated detection of seven software performance anti-patterns in Java-based applications was presented. Such framework exploits both static and dynamic information of software systems, thus assessing its accuracy in the detection of the anti-patterns. In the literature, many approaches have been defined to detect code smells [10], and several investigations are performed, e.g., the analysis of (i) inter-smell interactions to understand their impact [11] and (ii) sequences of different types of bad smells to improve both the detection and the solution [12]. Static and dynamic approaches differ from detecting performance issues at different stages, i.e., during the development and operational phases, respectively. The trade-off is between effort costs and accuracy, given that design changes are usually cheaper, whereas runtime information can probably contribute to capture a larger set of performance issues [13].

3) Data analysis of performance testing results: In [14] the authors developed an automated process for software performance anti-pattern detection that was based on data derived from performance testing and profiling data. In [15] a methodology was introduced for Software Performance Anti-pattern (SPA) characterization and detection, which applied SPA statistical characterization and multivariate analysis based on performance testing results. The methodology was applied to a large telecom system and was able to automatically identify top five software bottleneck services.

B. Architecture Anti-Patterns

Researchers have proposed various design and architecture smell definitions and detection techniques [16, 17, 18, 19]. Mo *et al.* [20] introduce a set of high-level issues called architecture anti-patterns, empirically shown to indicate excessive maintenance costs. Defined through Baldwin and Clark's design rule theory [21] and established design principles, these anti-patterns aim to detect influential abstractions that can decouple systems into independent modules. Researchers study anti-patterns due to their proven applicability in industry and their use of co-change history, which helps identify design debt with real maintenance penalties [22, 23, 24, 25, 26]. DV8, the supporting tool, assesses the maintenance cost of each anti-pattern instance by file count, file percentage, churn, and churn percentage, allowing for severity assessment. These anti-patterns leverage co-change history, which helps identify real design debt with added maintenance costs [27]. The DV8 [28] tool reports each anti-pattern's maintenance impact by file count, file percentage, churn, and churn percentage, enabling severity assessment.

Researchers have proposed a suite of anti-patterns specific to microservices [29, 30, 31]. In particular, Taibi *et al.* [29, 30] summarized a taxonomy of microservices, including technical and organizational anti-patterns. Technical anti-patterns are categorized into two main types:

- Internal anti-patterns: Primarily focus on implementation details, such as API versioning.
- Communication anti-patterns: Most relevant to this work, including Cyclic Dependency, ESB Usage, No API-Gateway, Shared Libraries, and Timeout.

These anti-patterns currently lack a unified representation, making it unclear which are most correlated with performance anti-patterns.

Fang *et al.* [32] first presented an approach that extracts and aggregates different types of dependencies into one design structure matrix model, and defined additional types of anti-patterns concerning distributed systems in general, including retiring components, data coupling, unstable API, etc. In this paper, we use this unified model to visualize multiple types of architecture anti-patterns and analyze their relations with performance anti-patterns.

C. Correlation between Performance and Architecture Anti-patterns

In [33] the authors model and analyze seven design patterns that industrial practitioners indicate as relevant for the performance of microservice systems. Table I briefly reports the main findings of the investigated architectural patterns, so that software architects are informed of possible system bottlenecks. A follow-up work [34] assesses the performance evaluation of three design patterns (i.e., *Gateway Aggregation*, *Gateway offloading*, and *Pipes and Filters*) by building an experimental environment and collecting real performance measurements. Overall, the experimental results confirm the performance variations observed by the previous theoretical performance estimations, even if the absolute values of real measurements w.r.t. model-based predictions can deviate. Such a deviation can be motivated by the minimal/maximal load to which the system is exposed, or the requests' heterogeneity that impacts on a different utilization of the system resources.

TABLE I
UNDERSTANDING THE CORRELATION BETWEEN ARCHITECTURAL PATTERNS AND SYSTEM PERFORMANCE [33]

Architectural Pattern	Performance Implications
Anti-corruption Layer	An adapter manages requests between different microservices. This can result in performance. pain due to the overhead of translating requests
Backends for Frontends	It avoids customizing a single backend microservice for multiple interfaces. This is a possible performance gain since it improves the autonomy and flexibility of the management w.r.t. generalized backends.
Command and Query Responsibility Segregation (CQRS)	It separates read and update operations for a data store microservice. This can be a performance gain since it speeds up read and write requests in case of no data overlap.
Gateway Aggregation	A gateway aggregates requests to multiple microservices in a single inquiry. This can result in a performance gain since communication overhead is reduced.
Gateway Offloading	It offloads common functionalities of multiple microservices to a proxy gateway. This is a performance gain since some tasks are offloaded to the gateway.
Pipes and Filters	It decomposes a complex task (i.e., single service) into separate elements (i.e., microservices) that can be independently managed. This might represent a performance gain due to the parallel computing of separate components.
Static Content Hosting	It deploys static content into cloud-based storage microservice. This is perceived as a performance gain since some requests are managed more rapidly.

III. SCALABILITY ASSESSMENT FRAMEWORK

In this section, we describe the **performance and scalability assessment framework (PSAF)** we used to identify architecture components responsible for performance and scalability deviations. The PSAF is illustrated in Figure 1.

The following dimensions were selected to build an end-to-end PSAF:

1) **Service Level Metric Objective (SLMO)** – response time and throughput,
2) **High-Level Model (HLM)** – queuing-theoretic analytical, simulation model, parametrization from measurement of the production environment, test cases,
3) **Operational Profile Definition (OPD)** – workload trace and automated analysis of historical data, probability of occurrence of load levels,
4) **Automated Load Test Generation (ATCG)** – operational profile sampling,
5) **Software Architecture Assessment (SAS)** – evaluates domain metric,
6) **Identify Architecture Component (IAC)** – performance deviations in experiments with variations.

Figure 1 illustrates the three key building blocks of the performance and scalability assessment framework: 1) Test design framework, 2) Measurement framework, and 3) Quality assessment framework.

The **test design framework** consists of the high-level model (HLM), operational profile definition (OPD), and the automated test case generation approach (ATCG). Parametrization from measurement of the production environment is used as input to HLM, and workload trace and analysis of historical data were selected as input to the OPD. In addition, the probability of the occurrence of load levels is computed by the HLM and is also used as input to the OPD. The ATCG uses as input sampling from the OPD. The HLM can also be used as an emulator to generate emulated test cases.

The **measurement framework** produces as output the performance and scalability assessment metrics: response time and throughput.

The **assessment framework** evaluates the domain-metric derived from empirical measurements to detect performance deviations from the service level metric objective. These performance deviations are used in the **quality assessment feedback process** to identify microservice architecture components that could be the root cause of these performance deviations.

IV. PERFORMANCE ANTI-PATTERNS

The following performance anti-patterns [35, 36, 37] were selected to investigate the correlation between architecture (described in Section V) and performance anti-patterns:

- *Application Hiccups* – a transient increase in application response time, followed by an equivalent decrease, is termed an 'application hiccup.' This behavior can be triggered when periodic tasks (e.g., garbage collection) are activated.
- *Continuous Violated Requirements* – can occur as a result of the inefficient implementation of a critical routine or third party software.
- *Traffic Jam* – can occur when a software bottleneck triggers a job queue causing persistent response time variability.
- *The Stifle* – manifests as the result of the execution of multiple similar database queries.
- *Expensive Database Call* – manifests as the result of the execution of one long response time database query.
- *Empty Semi Trucks* – similar to *the Stiffle* it manifests as the result of the execution of several similar shorts requests, causing inefficient use of interface resources.
- *The Blob* – manifests when one god class is implemented to either execute most of the required work, or to hold most of the data.

V. DV8 ARCHITECTURE ANTI-PATTERNS DETECTION

In DV8, anti-patterns can be visualized using a *Design Structure Matrix* (DSM). The columns and rows of this square matrix are labeled with the same set of elements, that is, files, packages, or service components, in the same order, and a marked cell indicates that the row element *depends* on the column element due to certain relations. For example, the ``C2, E'' in the cell at the intersection of the 5th row and 18th column in Figure 5 means the `seat` service calls the `order` service 2 times, and they share one entity. The number on the first row, first column, and the diagonal are the indexes of each element.

The key feature of DV8 is that it can integrate different dependency types. By default, DV8 can integrate static dependencies, e.g., call, inherit, with evolution couplings, that is, co-change relations among files, and detect the following history-based anti-patterns: (1) *Unstable Interface/API* (UIF): an element with a large number of dependent elements that also changes frequently with those dependent files; (2) *Modularity Violation Group* (MVG): a group

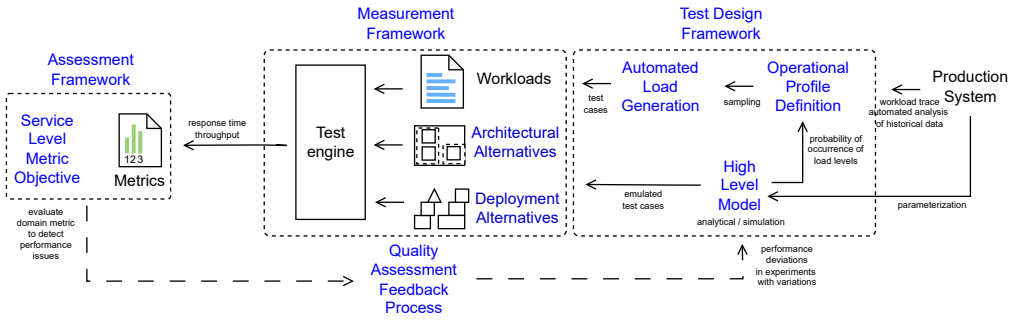


Fig. 1. Performance and scalability assessment framework

of elements that frequently co-change with structurally unrelated peers. The higher the co-change frequency, the stronger the coupling; and (3) *Crossing* (CRS) or *Crossing API*: a file or service with a large number of dependent and dependee files that also changes frequently with those dependent and dependee files.

When the co-change information is not available, DV8 can still detect the following anti-patterns: (1) *Unhealthy Inheritance Hierarchy* (UIH): a group of files with one of the following two structures: a parent class depends on its sub-classes, or the client of the hierarchy uses both the parent and the children, which are empirically validated to be high-maintenance [38, 20, 23, 39, 40, 24]; (2) *Clique* (CLQ): a set of files or services where each file directly or indirectly depends on every other file, and the whole file group forms a strongly connected graph. In Figure 5, services from 4 to 24 either call each other or share entities, and thus form a clique through these two relations; (3) *Package Cycle* (PKC): a pair of mutually dependent packages, which is not applicable in this paper. In addition, DV8 also displays the fan-in and fan-out of each element so that a god class or an overly complicated element can be easily detected.

To assess the heterogeneous types of dependencies within a microservice system, in this study, we combined the call relation and the entity-sharing relation among service components into one DSM. Figure 5 is an example showing all the service components involved in the scenarios shown in Figure 4. We manually recovered the call relation and entity sharing relation among these components, represented these two types of dependencies into a unified JSON format, and merged them into one DSM using DV8.

VI. Identification of Performance Bottlenecks

The **performance and scalability assessment** framework, described in Section III, compares the performance and scalability assessment metrics with the service level metric objective to detect performance issues. These detected performance issues are used by the HLM to detect architecture components that are the root cause of performance issues. The identification of architecture components that are the root cause of performance problems follows the approaches recommended in [37, 41]. In [37] the Normalized Distance and slope of the performance requirement are used to identify performance anti-patterns. In [41] response time and throughput are tracked as a function of load to identify resource bottlenecks.

The following process is used to identify bottlenecks and performance anti-patterns:

1) **Baseline architecture**: use architecture analysis per approach in [32] to reconstruct the communication structure between and within microservices.
2) **Baseline scalability requirement**: execute the low load performance tests to derive the scalability requirement: $R = \bar{x} + 3\sigma$ measured at low load.
3) **Baseline performance**: execute performance tests with increased load to compute the Normalized Distance (ND) of the scalability requirement, $2\frac{T}{T+R}$, where T is the measured response time, and R is the scalability requirement computed in Step 2.
4) **Identify degrading microservices**: compute the ND, and average response time slope, as described in [37].
5) **Correlate between architecture and performance microservices identified as problematic**: compare microservices marked as degrading with high load from the performance perspective in Step 3, with the microservices detected as problematic in the baseline architecture of Step 1.

VII. Empirical Evaluation of the Scalability Framework

In this section we present the empirical evaluation of this new framework in a production-like environment, to assess the correlation between architecture anti-patterns and performance anti-patterns. The approach consists of the following six steps, as illustrated in Figure 2.

1) Reconstruct dependencies using DV8: the tool DV8 is used to identify dependencies based on static source code analysis. The result shows that, if only considering static relations, these services appear to be independent of each other, as expected.
2) Parse microservice-to-microservice calls: Next we identify inter-microservice calls by determining when each microservice establishes HTTP connections to other services, and how these services share data elements among themselves.
3) Identification of architecture anti-patterns: based on the results of the previous steps, through manual inspection, identify microservices with potential anti-patterns, e.g., high coupling.
4) Performance tests of endpoints: as described below, we run performance tests with increasing loads, which identify endpoints with increasing response times.

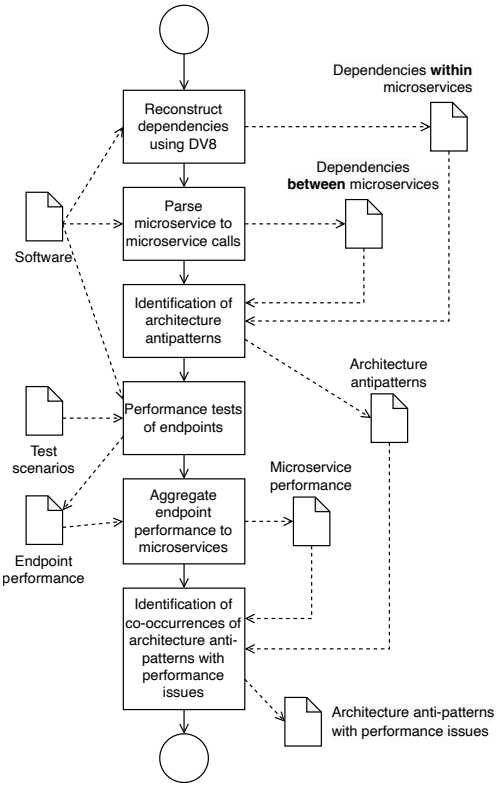


Fig. 2. Empirical evaluation approach combining architecture and performance analysis

5) Aggregate endpoint performance to microservices: we manually associate each endpoint with its respective microservice to evaluate the overall performance of each microservice.
6) Identification of co-occurrences of architecture antipatterns with performance issues: we compare the architecture of a microservice (including its communication structure to other microservices) with their performance behavior under increasing load.

The following section introduces the Train Ticket benchmark microservice in detail.

A. Train Ticket

Train Ticket is a containerized, microservice-based web application benchmark, composed of over 40 microservices implemented using modern technology stacks, as described in [42]. In this paper, we use a forked version of the original Train Ticket repository[1] since the original version seems not to be under development anymore.

A typical interaction flow within the Train Ticket system would execute login, booking a ticket booking and payment, and using or canceling the ticket. Each functionality triggers a sequence of interactions between the different microservices that compose the Train Ticket system. Microservices in Train Ticket do interact with each other and are organized into different layers depending on the dependencies among microservices. A microservice A is dependent on microservice B if A sends a request to B. Figure 3 depicts the architecture of the Train Ticket system in the form of a call graph.

For our experiments with the Train Ticket system, we have defined the workload specification as a set of requests that corresponds to expected user behavior when interacting with the Train Ticket system (the system under test). This behavior considers: logging in with the administrator and creating a user; logging in with the user with token generation and accessing the home page; getting trip information for departure and return data; selecting the trip and performing the booking steps: get contact info, get insurance, food and consign options; ticket payment (if status is booked); and collecting and using the ticket. Table II describes the requests sent to the Train Ticket system that corresponds to each user action. It defines the API endpoint and arguments.

In addition, Figure 4 illustrates a sequence diagram depicting the interactions between the microservices for ticket booking. It illustrated service dependencies, data flow, and the order of operations.

Our performance evaluation consisted of testing the endpoints described in Table II. We utilized the PPTAM[2] (Production and Performance Testing Application Monitoring) tool ecosystem [43]. The setup involved a containerized deployment of the Train Ticket system on Kubernetes, managed locally using Minikube. Minikube creates a local, single-node Kubernetes cluster with all essential Kubernetes components, including the API server, controller manager, and scheduler, to provide a complete Kubernetes environment.

In this setup, each microservice and database of the Train Ticket system was deployed as a separate Kubernetes pod, allowing isolated runtime environments and better resource allocation across components. This architecture enabled us to simulate realistic interactions within a microservices environment, where each service instance and the database operated independently while communicating over the network.

The experimental setup included two primary components: the *Drive* component (PPTAM) operated on a Linux virtual machine with 8 GB RAM and 4 logical CPUs at 2.095 GHz per core, while the *Testbed* component (Train Ticket system) was deployed in the Minikube environment on a Linux virtual machine with 64 GB RAM and 8 logical CPUs at 2.095 GHz per core.

The approach used in this performance evaluation consists of the following steps:

Experiment Generation In this step, we define the experiment settings for the test case. The test cases consist of a series of experiments, each performed according to a load test specification and evaluated against a baseline requirement [43]. Overall, the experiment setting aims to simulate an user interaction flow while capturing detailed performance metrics, with failure handling and retry logic ensuring robustness in the test environment. The load test specification includes the workload situation, which refers to the automated generation of the operations described in Table II, as well as the definition of the number of concurrent users. In our test case, we conducted experiments

[1]https://github.com/CUHK-SE-Group/train-ticket/

[2]PPTAM is an open-source software project available at: https://github.com/pptam/pptam-tool.

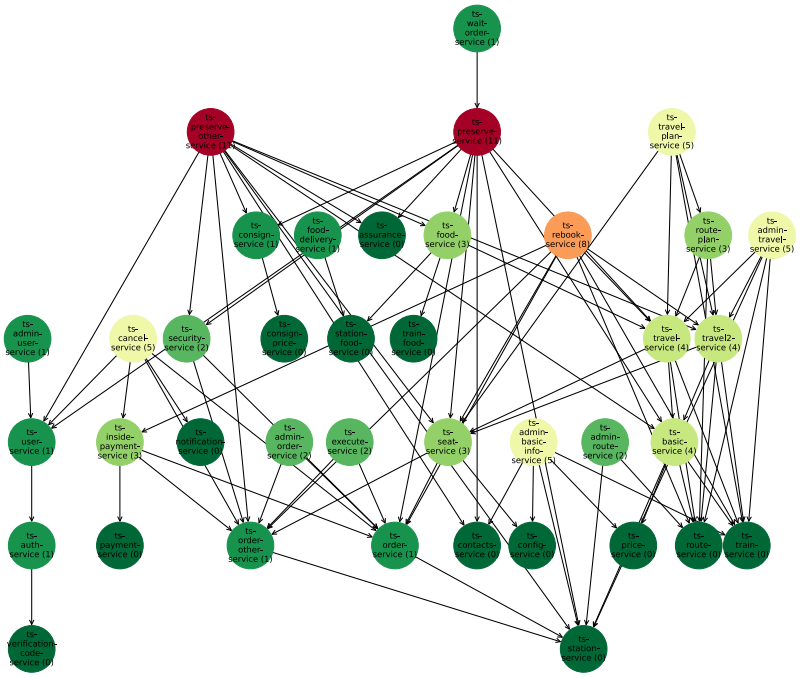


Fig. 3. Train Ticket call graph

TABLE II
FUNCTIONALITIES, API ENDPOINTS AND PARAMETERS FOR TRAIN TICKET BENCHMARK.

Functionality	Service	Relative path from /api/v1	Method	Arguments
User login	Users	/users/login	POST	username: string, password: string
Create contact	Contacts	/contactsservice/contacts	POST	name: string, accountId: string, documentType: int, "documentNumber: string, phoneNumber: string
Get trip information	Travel	/travelservice/trips/left	POST	startPlace: string, endPlace: string, departureTime: string
Get contacts	Contacts	/contactsservice/contacts/account/{user_id}	GET	—
Get foods	Food	/foodservice/foods/{date}/{startStation}/{endStation}/{tripId}	GET	—
Get insurance types	Assurance	/assuranceservice/assurances/types	GET	—
Book ticket	Preserve	/preserveservice/preserve	POST	accountId: string, contactsId: string, tripId: string, seatType: int, date: string, from: "string, to: string, assurance: int, foodType: int, foodName: string, foodPrice: double, stationName: string, storeName: string
Get last order	Order	/orderservice/order/refresh	POST	loginId: string, enableStateQuery: boolean, enableTravelDateQuery: boolean, enableBoughtDateQuery: boolean, travelDateStart: string, travelDateEnd: string, boughtDateStart: string, boughtDateEnd: string
Pay ticket	Payment	/inside_pay_service/inside_payment	POST	orderId: string, tripId: string
Colletc ticket	Execute	/executeservice/execute/collected/{orderId}	GET	—
Use ticket in station	Execute	/executeservice/execute/execute/{orderId}	GET	—

with 1, 5, 10, 15, 20, and 25 concurrent users. The workload scenario corresponding to 1 user serves as the baseline workload for identifying the baseline requirement. Each test ran for 20 minutes, resulting in a total testing time of 120 minutes.

Experiment execution In this step, each endpoint is tested under the experiment settings defined in the previous step. For each endpoint, we collected a set of performance metrics, which will serve as the basis for calculating the Mean Normalized Distance for each endpoint/microservice (the results are analyzed in section VII-B). The performance results are presented in Table III, where we provide the average response time ($\mathbb{E}[T]$), maximum response time ($T_{\max}$), and number of failures ($N(F)$). For the sake of conciseness, in this table we report results for Test 1 (one user), Test 3 (ten users), and Test 5 (twenty users). We report results accounting for all tests in Figures 6 and 7.

B. Empirical Results

1) Reconstruction dependencies using DV8: DV8 was used to analyze the Train Ticket system and to create the initial design structure matrix. When we only consider static relations among these service components, they appear to be independent of one another, which is expected because

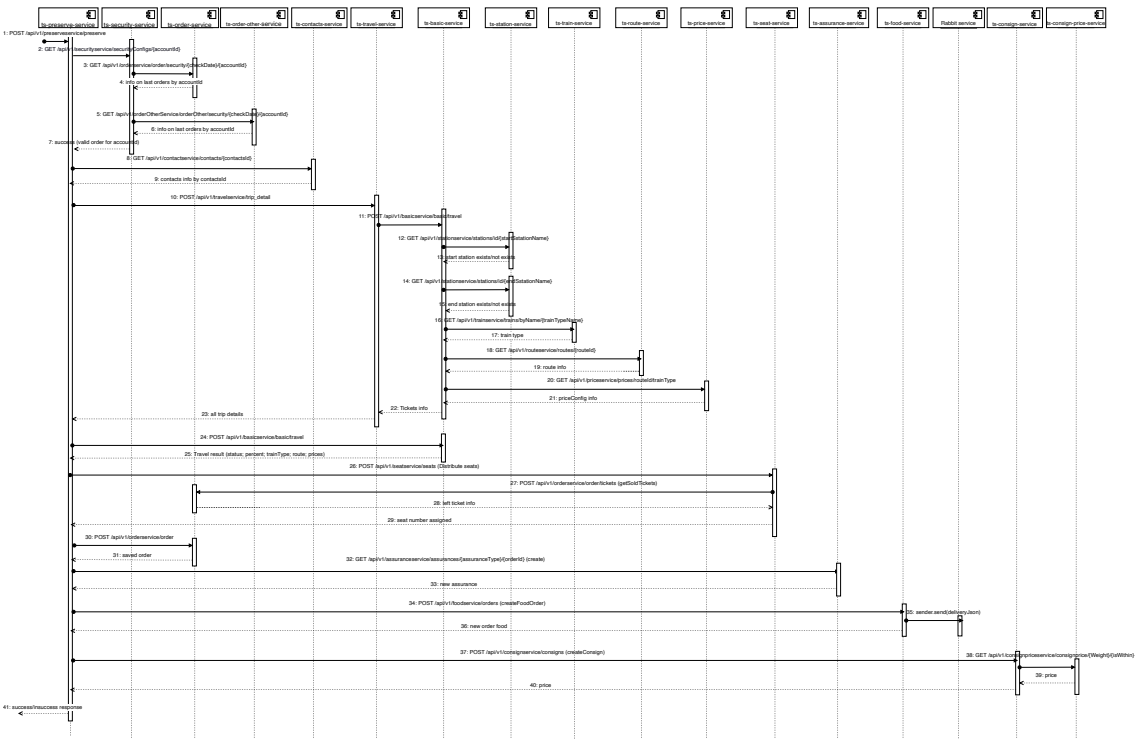


Fig. 4. A fragment of Sequence Diagram of the Book a Ticket scenario (POST /api/v1/preserveservice/preserve API call).

TABLE III
TRAIN TICKET PERFORMANCE RESULTS.

Relative path from /api/v1	Test 1 (1 user)			Test 3 (10 users)			Test 5 (20 users)		
	$\mathbb{E}[T]$ (ms)	$T_{\max}$ (ms)	$N(F)$	$\mathbb{E}[T]$ (ms)	$T_{\max}$ (ms)	$N(F)$	$\mathbb{E}[T]$ (ms)	$T_{\max}$ (ms)	$N(F)$
/users/login	130.57	153.55	0	177.35	243.91	0	254.81	895.99	0
/contactservice/contacts	49.33	49.33	0	31.61	54.25	0	34.21	96.42	0
/travelservice/trips/left	128.92	210.43	0	138.63	617.18	0	8864.92	349530.81	9
/contactservice/contacts/account/{user_id}	11.75	36.69	0	9.44	52.25	0	12.69	37.87	0
/foodservice/foods/{date}/ {startStation}/{endStation}/{tripId}	43.43	109.60	0	32.49	98.30	0	597.57	22289.25	0
/assuranceservice/assurances/types	9.32	29.19	0	8.00	81.40	0	10.94	62.34	0
/preserveservice/preserve	344.33	495.94	0	685.40	2699.25	0	42331.99	376227.90	34
/orderservice/order/refresh	14.49	52.30	0	20.45	292.80	0	1146.92	23150.74	0
/inside_pay_service/inside_payment	52.58	93.33	0	70.29	314.56	0	3481.73	334729.73	1
/executeservice/execute/collected/{orderId}	22.03	42.94	0	27.80	338.88	0	3707.63	335011.35	1
executeservice/execute/execute/{orderId}	21.76	48.85	0	27.36	313.79	0	5509.01	337643.53	2

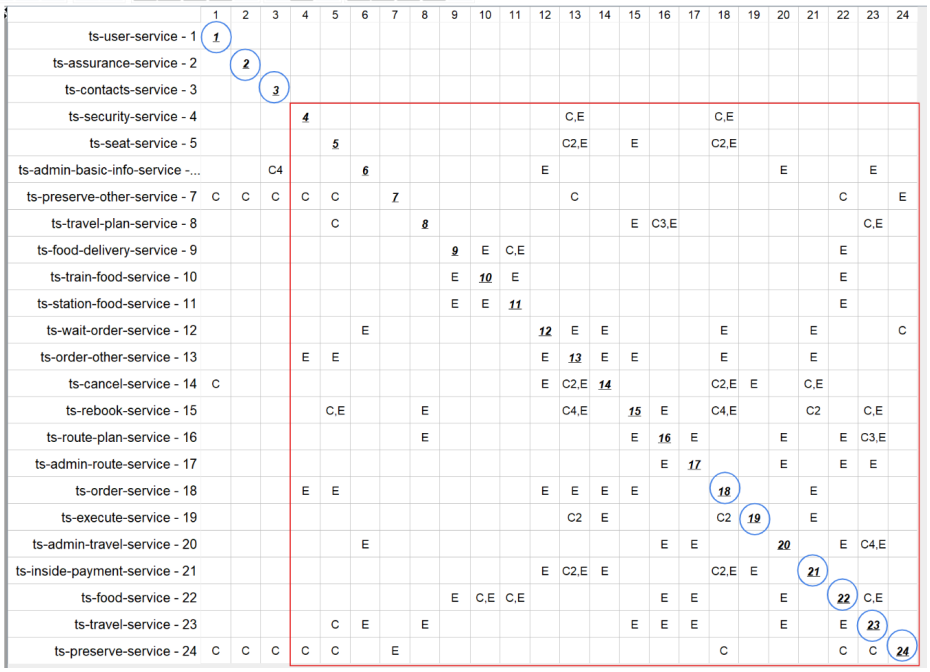
these service components are containerized. But they are not actually independent, as we will elaborate next.

2) Parse microservice-to-microservice calls: The analysis of the software under test resulted in the identification of the microservice-to-microservice call graph (see Fig. 3).

Recall that the call graph was parsed from the source code identifying all places where microservices opened HTTP calls to other microservices. In the graph, the number of distinct microservices called by a specific microservice is indicated in brackets. Microservices calling many distinctive other microservices are depicted in red and gradually towards green if they have fewer connections to other services. The result of this step was imported by DV8 to create a combined DSM as shown in Fig. 5.

Relationship between call graph, sequence diagram, and DSM. Figure 3 (call graph) and Figure 5 (DSM) were generated semi-automatically and report a global perspective towards the system, while Figure 4 (sequence graph) was produced manually and represents an example of local interaction flows. To produce Figure 4 we chose the "preserve" endpoint because it was the API call that showed more performance problems, in addition to presenting high out-degree in Figure 3.

There is a clear relationship between Figures 3 and 5: an edge $x \rightarrow y$ in Figure 3 corresponds to a C entry in cell (x, y) of Figure 5, reflecting a coupling relationship derived from the call graph. In addition, interactions shown in Figure 4 also correspond to C entries in the DSM.



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
ts-user-service - 1	1																							
ts-assurance-service - 2		2																						
ts-contacts-service - 3			3																					
ts-security-service - 4				4									C,E					C,E						
ts-seat-service - 5					5								C2,E		E			C2,E						
ts-admin-basic-info-service -...			C4			6						E								E			E	
ts-preserve-other-service - 7	C	C	C	C	C		7						C									C		E
ts-travel-plan-service - 8					C			8							E	C3,E							C,E	
ts-food-delivery-service - 9									9	E	C,E											E		
ts-train-food-service - 10									E	10	E											E		
ts-station-food-service - 11									E	E	11											E		
ts-wait-order-service - 12						E						12	E	E				E			E			C
ts-order-other-service - 13				E	E							E	13	E	E			E			E			
ts-cancel-service - 14	C											E	C2,E	14				C2,E	E		C,E			
ts-rebook-service - 15					C,E			E					C4,E		15	E		C4,E			C2		C,E	
ts-route-plan-service - 16								E							E	16	E			E		E	C3,E	
ts-admin-route-service - 17																E	17			E		E	E	
ts-order-service - 18				E	E							E	E	E	E			18			E			
ts-execute-service - 19													C2	E				C2	19		E			
ts-admin-travel-service - 20						E										E	E			20		E	C4,E	
ts-inside-payment-service - 21												E	C2,E	E				C2,E	E		21			
ts-food-service - 22									E	C,E	C,E					E	E			E		22	C,E	
ts-travel-service - 23					C	E		E							E	E	E			E		E	23	
ts-preserve-service - 24	C	C	C	C	C		E											C				C	C	24

Fig. 5. DV8 output

Conversely, E entries in Figure 5 have no direct counterpart in Figure 3. Instead, parameter passing depicted in Figure 4 may suggest an E relationship in Figure 5. Determining precise rules to link E entries in Figure 5 to specific interactions in Figure 4 remains an open challenge, as noted in Section VIII.

3) Identification of Architecture Anti-Patterns: To identify architecture anti-patterns, we first extracted the call relations (labeled using 'C') among service, and analyzed their data coupling by investigating if two services share the same entities (labeled using 'E'). Figure 5 displays all the services involved in the Book a Ticket Scenario and their relations. From this DSM, we can identify a clique formed by 21 services, which either call each other or share entities. The services with circles are those listed in Table III. The DSM revealed that the following three services are independent of other services: `user`, `assurance`, and `contacts`. All other circled services are involved in the clique. In particular, several services depend on a large number of other services, such as `Preserve`, `travel`, and `order`.

4) Performance Tests of Endpoints: After performing the experiments described in Section VII-A we obtained the average and maximum response times, the standard deviation and the number of failures. The results for the tests 1, 3, and 5 (1, 10, and 20 users) are reported in table III. Figure 6 reports the Normalized Distance (calculated as described in Sect. VI) for each experiment and endpoint. Analyzing these results, we see that the endpoints marked as ④, ⑤, ⑥, ⑦, ⑧, ⑨, and ⑩ show significant performance degradation when load increases, i.e., reach a value near 2, while the endpoint marked as ① shows minor performance degradation, i.e., reach a value of 1.06.

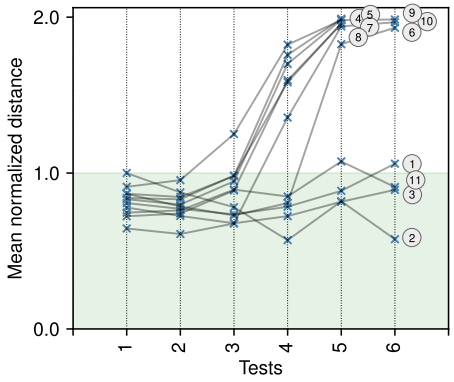


Fig. 6. Normalized distance

To further study the collected data, we calculated the slope [37], which describes how much the average response time changes over the range of utilization. Figure 7 visualizes the results for each endpoint, positioning it on the horizontal axis on the Normalized Distance it achieved in the last experiment, and on the vertical axis at the slope it reached. For example, endpoint number ⑨ (`/api/v1/preserveservice/preserve`), in the last experiment, shows significant violation of the defined performance requirement and the steepest degradation rate, as shown by the slope of response time. Both outliers shown in Fig. 7 ⑨ and ⑩ *also* are two endpoints belonging to microservices with the highest Coupling Values (see Table V).

Table V illustrates the achieved Normalized Distance values for each endpoint, the associated microservice, and

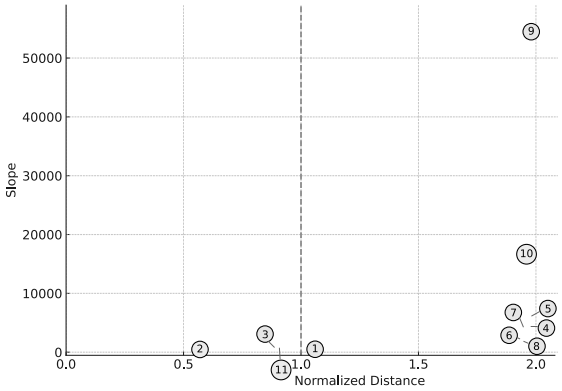


Fig. 7. Slope vs Normalized Distance

TABLE IV
LINKAGE OF ENDPOINTS TO MICROSERVICES

API Endpoint	Microservice
/api/v1/executeservice/execute/execute/-order	ts-execute-service
/api/v1/preserveservice/preserve	ts-preserve-service
/api/v1/executeservice/execute/collected/-order	ts-execute-service
/api/v1/travelservice/trips/left	ts-travel-service
/api/v1/inside_pay_service/inside_payment	ts-inside-payment-service
/api/v1/orderservice/order/refresh	ts-order-service
/api/v1/foodservice/foods/departure/-shanghai/suzhou/D1345	ts-food-service
/api/v1/assuranceservice/assurances/types	ts-assurance-service
/api/v1/users/login	ts-user-service
/api/v1/contactservice/contacts/account/-user	ts-contacts-service
/api/v1/contactservice/contacts	ts-contacts-service

the Coupling Value of each microservice.

We can observe that the microservice (`ts-preserve-service`) implements a Blob (god class) performance anti-pattern, because it controls the full functionality and issues several data requests to other microservices. This microservice had the largest slope and Normalized Distance assessment in the performance analysis and is also part of a clique architecture anti-pattern identified by DV8.

5) Aggregate Endpoint Performance to Microservices: In this step, we linked each endpoint to a microservice as shown in Table IV.

6) Identification of the Correlation of Microservices Coupling with Performance Issues: To analyze the relationship between the **Normalized Distance** of endpoints (which increases as the performance of microservices degrades) and the **Coupling Value** (also referred to as the node degree in the call graph) per microservice, we computed multiple metrics, including Pearson correlation, Spearman correlation, Normalized Mutual Information (NMI), and Cosine similarity. For completeness, we briefly introduce the considered metrics and measures in Appendix A. The data used to compute the corelation metrics and similarity measures is reported in Table V (see columns titled **Normalized Distance** and **Coupling Value**).

The results of these calculations highlight key insights into the relationship between Normalized Distance and coupling. The **Pearson correlation coefficient**, valued at 0.555, indicates a moderate positive linear correlation, reflecting a general proportionality between the two variables. The **Spearman correlation coefficient** is higher, at 0.734, revealing a stronger monotonic relationship, suggesting that as one variable increases, the other tends to do so consistently, even if not linearly. The **Normalized Mutual Information (NMI)**, with a value of 0.769, quantifies the shared information between the variables, underscoring a significant connection. Finally, the **Cosine similarity**, calculated at 0.740, demonstrates a strong directional alignment between the vectors, suggesting that Normalized Distance and coupling share similar patterns despite differences in magnitude.

The clique anti-pattern correlates with poor performance. Services not involved in the Clique anti-pattern, such as `user`, `assurance`, and `contacts`, exhibit minimum response times in the range of tens to hundreds of milliseconds. In contrast, services affected by the Clique demonstrate response times thousands of times greater, highlighting significantly degraded performance. This trend is further confirmed in Table V. Among the four elements with the lowest Normalized Distance – indicating higher performance – three, namely 1, 2, and 3, are those not part of the Clique anti-pattern, providing additional evidence of the relationship between architectural design and system performance.

Takeaway message. Our findings suggest that as the coupling of microservices increases, there is a tendency for their Normalized Distance to increase, supporting the hypothesis that architectural metrics, e.g., as captured by the Coupling Values, are correlated with performance deviations, e.g., as captured by Normalized Distance. Another observation is that services involved in a Clique anti-pattern perform significantly worse.

VIII. THREATS TO VALIDITY

In this work we implemented an empirical assessment to identify correlations between performance and architecture issues. The following threats to validity were identified.

Internal validity. This is an initial research that used only one benchmark. Even tough our empirical assessment identified correlations between performance and architecture issues, the results might be affected by the lack of diversity in benchmarks and deployment architectures. We are currently extending our research by incorporating additional benchmarks.

Construct validity. We plan to implement an automated approach to derive correlations between performance and architecture anti-patterns. The assessment presented in this paper was based on expert assessment, which is difficult to generalize.

External validity. The applicability of our results to a more general context is limited by gaps in the automated environment used. There is a need to integrate PPTAM and DV8 with automated performance anti-pattern detection tools to overcome this limitation.

TABLE V
ENDPOINTS, NORMALIZED DISTANCE, THE ASSOCIATED MICROSERVICE, AND THE RELATED COUPLING

Number	API Endpoint	Normalized Distance	Microservice	Coupling Value
⑤	/api/v1/executeservice/execute/execute/order	1.98	ts-execute-service	2
⑨	/api/v1/preserveservice/preserve	1.98	ts-preserve-service	11
④	/api/v1/executeservice/execute/collected/order	1.98	ts-execute-service	2
⑩	/api/v1/travelservice/trips/left	1.96	ts-travel-service	4
⑦	/api/v1/inside_pay_service/inside_payment	1.95	ts-inside-payment-service	3
⑧	/api/v1/orderservice/order/refresh	1.95	ts-order-service	1
⑥	/api/v1/foodservice/foods/departure/shanghai/suzhou/D1345	1.93	ts-food-service	3
①	/api/v1/assuranceservice/assurances/types	1.06	ts-assurance-service	0
⑪	/api/v1/users/login	0.91	ts-user-service	1
③	/api/v1/contactservice/contacts/account/user	0.89	ts-contacts-service	0
②	/api/v1/contactservice/contacts	0.57	ts-contacts-service	0

TABLE VI
RELATION BETWEEN NORMALIZED DISTANCE AND COUPLING VALUE

Metric	Value	Range of Possible Values	
		Minimum	Maximum
Pearson Correlation	0.555	-1.0	1.0
Spearman Correlation	0.734	-1.0	1.0
Normalized Mutual Information	0.769	0.0	1.0
Cosine Similarity	0.740	-1.0	1.0

IX. CONCLUSION AND FUTURE WORK

Microservice architecture design must address stakeholder needs for maintainability, scalability, and availability, requiring a dynamic assessment process for detecting performance deviations and identifying root causes. While performance anti-patterns correspond to dynamic metric deviations, architecture anti-patterns expose structural issues by identifying misalignments with formal design standards.

This study presents an assessment using DV8 and PPTAM tools on the Train Ticket benchmark, showing correlations between performance and architecture issues. Specifically, we have found that one microservice (`ts-preserve-service`) implements a Blob (god class) performance anti-pattern. This microservice had the largest slope and Normalized Distance assessment in the performance analysis and is also part of a clique architecture anti-pattern identified by DV8. This result shows that quantitative performance analysis can guide architecture refactoring prioritization efforts. We used node degree in call graphs as a network centrality metric, revealing a correlation between such architecture metric and performance metrics, such as Normalized Distance. Our findings suggest that high-degree nodes, which often result from a lack of attention to modularity by developers, represent an architectural anti-pattern. This anti-pattern is shown to correlate with reduced performance, as indicated by higher Normalized Distance values. In future work, we plan to experiment with architecture-inspired performance improvements and further explore the relationship between architecture and performance by considering additional network centrality metrics, such as PageRank, betweenness, and closeness, while also relating these to broader performance metrics like throughput. This approach aims to deepen understanding of how architectural features impact performance, supporting the development of more efficient, scalable, and resilient microservice systems, and bridging the systems architecture and performance teams.

APPENDIX A
CORRELATION AND SIMILARITY MEASURES

Next, we briefly introduce the definitions of correlation metrics and similarity measures considered in our analysis.

Pearson Correlation. The Pearson correlation coefficient measures the linear relationship between two variables. It is defined as:

$$r = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2}\sqrt{\sum_{i=1}^{n}(y_i - \bar{y})^2}} = \frac{\operatorname{cov}(X, Y)}{\sigma_X \sigma_Y},$$

where x_i and y_i are the individual data points for the two variables (Normalized Distance and coupling), $\bar{x}$ and $\bar{y}$ are their respective means, and n is the number of data points. It ranges from -1 (perfect negative correlation) to +1 (perfect positive correlation) [44]. For our data, $r = 0.555$, indicating a moderate positive linear correlation.

Spearman Correlation. The Spearman correlation coefficient quantifies the strength of a monotonic relationship between two variables. It is defined as the Pearson correlation coefficient between the ranked variables:

$$\rho = \frac{\operatorname{cov}(\mathrm{R}[X], \mathrm{R}[Y])}{\sigma_{\mathrm{R}[X]} \sigma_{\mathrm{R}[Y]}},$$

where $\mathrm{R}[X]$ and $\mathrm{R}[Y]$ are the ranks of X and Y, respectively, $\operatorname{cov}(\mathrm{R}[X], \mathrm{R}[Y])$ is the covariance of the ranks, and $\sigma_{\mathrm{R}[X]}$ and $\sigma_{\mathrm{R}[Y]}$ are the standard deviations of the ranks.

When multiple values have the same rank they are assigned the average of the tied ranks. For example, in our data, 1.98 appears three times among 11 elements, and these tied values are assigned the average rank of 10.

The resulting Spearman correlation coefficient ranges from -1 (perfect negative monotonic relationship) to $+1$ (perfect positive monotonic relationship). For our data, the computed Spearman correlation coefficient is $\rho = 0.734$. This value suggests a strong monotonic relationship between Normalized Distance and Coupling Values, exceeding the linear relationship captured by the Pearson correlation.

Normalized Mutual Information (NMI). NMI quantifies the shared information between two variables:

$$\mathrm{NMI}(X, Y) = \frac{2 \cdot I(X; Y)}{H(X) + H(Y)},$$

where $I(X; Y)$ is the mutual information between X and Y, and $H(X)$ and $H(Y)$ are the entropies of X and Y,

respectively. NMI values range from 0 (no shared information) to 1 (complete information overlap). For our data, NMI $= 0.769$, highlighting a significant connection between Normalized Distance and coupling.

Cosine Similarity. Cosine similarity measures the cosine of the angle between two non-zero vectors, indicating their directional alignment:

$$\text{Cosine Similarity} = \frac{\sum_{i=1}^{n} x_i y_i}{\sqrt{\sum_{i=1}^{n} x_i^2} \sqrt{\sum_{i=1}^{n} y_i^2}},$$

where x_i and y_i are elements of the two vectors (Normalized Distance and coupling). The value ranges from -1 (exactly opposite), 0 (no similarity) to 1 (perfect similarity). For our data, Cosine Similarity $= 0.740$, indicating a strong directional alignment.

Acknowledgments

This work has been partially funded by the MUR-PRIN project 20228FT78M DREAM, MUR Department of Excellence 2023 - 2027 for GSSI, PNRR ECS00000041 VITALITY, FCT – Fundação para a Ciência e Tecnologia within the R&D Unit Project of ALGORITMI Centre, and the National Science Foundation of the US under grants CCF-2232720, CCF-2213764, and TI-2236824.

References

[1] Andre B. Bondi. Characteristics of scalability and their impact on performance. In *Second International Workshop on Software and Performance, WOSP 2000, Ottawa, Canada, September 17-20, 2000*, pages 195–203. ACM, 2000.

[2] Elaine J. Weyuker and Alberto Avritzer. A metric for predicting the performance of an application under a growing workload. *IBM Syst. J.*, 41(1):45–54, 2002.

[3] Ming Yan, XiaoMeng Liang, ZhiHui Lu, Jie Wu, and Wei Zhang. Hansel: Adaptive horizontal scaling of microservices using bi-lstm. *Applied Soft Computing*, 105:107216, 2021.

[4] Vittorio Cortellessa, Antinisca Di Marco, and Catia Trubiani. An approach for modeling and detecting software performance antipatterns based on first-order logics. *Software & Systems Modeling*, 13(1):391–432, 2014.

[5] Catia Trubiani, Anne Koziolek, Vittorio Cortellessa, and Ralf H. Reussner. Guilt-based handling of software performance antipatterns in palladio architectural models. *J. Syst. Softw.*, 95:141–165, 2014.

[6] Martina De Sanctis, Catia Trubiani, Vittorio Cortellessa, Antinisca Di Marco, and Mirko Flamminj. A model-driven approach to catch performance antipatterns in adl specifications. *Inf. Softw. Technol.*, 83(C):35–54, March 2017.

[7] Catia Trubiani, Achraf Ghabi, and Alexander Egyed. Exploiting traceability uncertainty between software architectural models and extra-functional results. *J. Syst. Softw.*, 125:15–34, 2017.

[8] Davide Arcelli, Vittorio Cortellessa, and Daniele Di Pompeo. Automating performance antipattern detection and software refactoring in uml models. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 639–643, 2019.

[9] Catia Trubiani, Riccardo Pinciroli, Andrea Biaggi, and Francesca Arcelli Fontana. Automated detection of software performance antipatterns in java-based applications. *IEEE Transactions on Software Engineering*, 49(4):2873–2891, 2023.

[10] Bartosz Walter, Francesca Arcelli Fontana, and Vincenzo Ferme. Code smells and their collocations: A large-scale experiment on open-source systems. *Journal of Systems and Software*, 144:1–21, 2018.

[11] Aiko Yamashita and Leon Moonen. Exploring the impact of inter-smell relations on software maintainability: An empirical study. In *International Conference on Software Engineering (ICSE)*, pages 682–691, 2013.

[12] Hui Liu, Zhiyi Ma, Weizhong Shao, and Zhendong Niu. Schedule of bad smell detection and resolution: A new way to save effort. *IEEE transactions on Software Engineering*, 38(1):220–235, 2011.

[13] Jinfu Chen, Weiyi Shang, and Emad Shihab. Perfjit: Test-level just-in-time prediction for performance regression introducing commits. *IEEE Transactions on Software Engineering*, 48(5):1529–1544, 2020.

[14] Catia Trubiani, Alexander Bran, André van Hoorn, Alberto Avritzer, and Holger Knoche. Exploiting load testing and profiling for performance antipattern detection. *Information and Software Technology*, 95:329–345, 2018.

[15] Alberto Avritzer, Ricardo Britto, Catia Trubiani, Barbara Russo, Andrea Janes, Matteo Camilli, André van Hoorn, Robert Heinrich, Martina Rapp, and Jörg Henß. A multivariate characterization and detection of software performance antipatterns. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering*, ICPE '21, page 61–72, New York, NY, USA, 2021. Association for Computing Machinery.

[16] Martin Lippert and Stephen Roock. *Refactoring in Large Software Projects: Performing Complex Restructurings Successfull*. Wiley, 2006.

[17] Joshua Garcia, Daniel Popescu, George Edwards, and Nenad Medvidovic. Toward a catalogue of architectural bad smells. In *Proceedings of the 5th International Conference on the Quality of Software Architectures: Architectures for Adaptive Software Systems*, pages 146–162, 2009.

[18] Joshua Garcia, Daniel Popescu, George Edwards, and Nenad Medvidovic. Identifying architectural bad smells. In *Proc. 13th*, pages 255–258, March 2009.

[19] Duc Le and Nenad Medvidovic. Architectural-based speculative analysis to predict bugs in a software system. In *Proceedings of the 38th International Conference on Software Engineering Companion*, pages 807–810, 2016.

[20] Ran Mo, Yuanfang Cai, Lu Xiao, Rick Kazman, and Qiong Feng. Architecture anti-patterns: Automatically detectable violations of design principles. *IEEE Transactions on Software Engineering*, 47(5), 2021.

[21] Carliss Y. Baldwin and Kim B. Clark. *Design Rules: The Power of Modularity Volume 1*. MIT Press, Cambridge, MA, USA, 1999.

[22] Ran Mo, Will Snipes Yuanfang Cai, , S. Ramaswamy, Rick Kazman, and Martin Naedele. Experiences applying automated architecture analysis tool suites.

In *Proceedings of the 33rd IEEE/ACM International Conference on Automated Software Engineering*, 2018.

[23] Rick Kazman, Yuanfang Cai, Ran Mo, Qiong Feng, Lu Xiao, Serge Haziyev, Volodymyr Fedak, and Andrey Shapochka. A case study in locating the architectural roots of technical debt. In *Proc. 37th*, May 2015.

[24] Wensheng Wu, Yuanfang Cai, Rick Kazman, Ran Mo, Zhipeng Liu, Rongbiao Chen, Yingan Ge, Weicai Liu, and Junhui Zhang. Software architecture measurement—experiences from a multinational company. In Carlos E. Cuesta, David Garlan, and Jennifer Pérez, editors, *Software Architecture*, pages 303–319. Springer International Publishing, 2018.

[25] M. Nayebi, Y. Cai, R. Kazman, G. Ruhe, Q. Feng, C. Carlson, and F. Chew. A longitudinal study of identifying and paying down architecture debt. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 171–180, 2019.

[26] Robert Schwanke, Lu Xiao, and Yuanfang Cai. Measuring architecture quality by structure plus history analysis. In *Proc. 35rd*, pages 891–900, May 2013.

[27] Jason Lefever, Yuanfang Cai, Humberto Cervantes, Rick Kazman, and Hongzhou Fang. On the lack of consensus among technical debt detection tools. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 121–130, 2021.

[28] Yuanfang Cai and Rick Kazman. Dv8: Automated architecture analysis tool suites. In *2019 IEEE/ACM International Conference on Technical Debt (TechDebt)*, pages 53–54, 2019.

[29] Davide Taibi, Valentina Lenarduzzi, and Claus Pahl. *Microservices Anti Patterns: A Taxonomy*. 06 2019.

[30] Davide Taibi and Valentina Lenarduzzi. On the definition of microservice bad smells. *IEEE Software*, 35(3):56–62, 2018.

[31] Mark Richards. *Microservices AntiPatterns and Pitfalls*. 07 2016.

[32] Hongzhou Fang, Yuanfang Cai, Rick Kazman, and Jason Lefever. Identifying anti-patterns in distributed systems with heterogeneous dependencies. In *2023 IEEE 20th International Conference on Software Architecture Companion (ICSA-C)*, pages 116–120, 2023.

[33] Riccardo Pinciroli, Aldeida Aleti, and Catia Trubiani. Performance modeling and analysis of design patterns for microservice systems. In *International Conference on Software Architecture (ICSA)*, pages 35–46, 2023.

[34] Willem Meijer, Catia Trubiani, and Aldeida Aleti. Experimental evaluation of architectural software performance design patterns in microservices. *J. Syst. Softw.*, 218:112183, 2024.

[35] Connie U. Smith and Lloyd G. Williams. Software performance antipatterns for identifying and correcting performance problems. In *International Computer Measurement Group Conference*, 2012.

[36] Alexander Wert. *Performance Problem Diagnosis by Systematic Experimentation*. PhD thesis, 2015.

[37] Alberto Avritzer, Ricardo Britto, Catia Trubiani, Matteo Camilli, Andrea Janes, Barbara Russo, André van Hoorn, Robert Heinrich, Martina Rapp, Jörg Henß, and Ram Kishan Chalawadi. Scalability testing automation using multivariate characterization and detection of software performance antipatterns. *Journal of Systems and Software*, 193:111446, 2022.

[38] Ran Mo, Yuanfang Cai, Rick Kazman, and Lu Xiao. Hotspot patterns: The formal definition and automatic detection of recurring high-maintenance architecture issues. In *Proc. 12th*, 2015.

[39] Ran Mo, Will Snipes, Yuanfang Cai, Srini Ramaswamy, Rick Kazman, and Martin Naedele. Experiences applying automated architecture analysis tool suites. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, page 779–789, New York, NY, USA, 2018. Association for Computing Machinery.

[40] M. Nayebi, Y. Cai, R. Kazman, G. Ruhe, Q. Feng, C. Carlson, and F. Chew. A longitudinal study of identifying and paying down architecture debt. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 2019.

[41] Riccardo Pinciroli, Aldeida Aleti, and Catia Trubiani. Performance modeling and analysis of design patterns for microservice systems. In *2023 IEEE 20th International Conference on Software Architecture (ICSA)*, pages 35–46, 2023.

[42] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chenjie Xu, Chao Ji, and Wenyun Zhao. Benchmarking microservice systems for software engineering research. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, pages 323–324, 2018.

[43] Alberto Avritzer, Vincenzo Ferme, Andrea Janes, Barbara Russo, Henning Schulz, and André van Hoorn. A quantitative approach for the assessment of microservice architecture deployment alternatives by automated performance testing. In *Software Architecture: 12th European Conference on Software Architecture, ECSA 2018, Madrid, Spain, September 24–28, 2018, Proceedings 12*, pages 159–174. Springer, 2018.

[44] David Freedman, Robert Pisani, and Roger Purves. Statistics (international student edition). *Pisani, R. Purves, 4th edn. WW Norton & Company, New York*, 2007.