ORIGINAL ARTICLE
Journal Section

Event-Sourced, Observable Software Architectures: an Experience Report

Francesco Alongi[1] | **Marcello M. Bersani**[1] | **Nicolò Ghielmetti**[1] | **Raffaela Mirandola**[1] | **Damian A. Tamburri**[2]

[1]Politecnico di Milano, Milano, Italy

[2]Eindhoven University of Technology - JADS, Eindhoven - s'Hertogenbosch, The Netherlands

Correspondence
Marcello M. Bersani, Politecnico di Milano, Italy
Email: marcellomaria.bersani@polimi.it

Funding information

The speeding growth of the IT market and the spreading of disruptive technologies are leading towards more and more risky operations in need of constant upkeep, monitoring as well as proactive orchestration. On the one hand, the property allowing a system to be catered by automated monitoring and healing technology is defined as *observability*. On the other hand, appropriate design principles to manifest observability were originally referred as *Event sourcing* by its inventor Martin Fowler and warrant for the aforementioned sustainable software operations.

Both Event sourcing and observability are complex to leverage on and design for. In an effort to understand more on both concepts, we offer an experience report on their practical use, featuring: (1) a rigorous definition of software architecture observability and a set of principles to design for observability using augmented forms of well-known design patterns in line with Event sourcing; and (2) an impact analysis in the context of a case study. Our Study reveals several interesting notions around the concept of observability but our findings also make explicit new architecture trade-offs that software architects and stakeholders need to consider as first-class architecture-level concerns.

KEYWORDS
Event Sourcing, Software Architecture Observability, Software Practice and Experience, Design Principles

1 | INTRODUCTION

Software systems nowadays are reaching an unprecedented size and scale with more and more stringent Quality-of-Service (QoS) requirements as well as service continuity conditions and costs [1]. At these magnitudes, monitoring and

discontinuity management in such stringent conditions become even more critical [2, 3, 4] and their effective provision demands proper tooling and code instrumentation. The portfolio of vendors selling off-the-shelf solutions is rather wide and the offer they provide includes various types of products. Software products for monitoring ranges from solutions providing few yet advanced functionalities (e.g., Logstash [5] for logging) to complex platforms supplying monitoring-as-a-service (e.g., Dynatrace [6], Splunk [7], Grafana Labs [8], Apache Skywalking [9]) that allow system administrators to identify problems, receive runtime alarms and monitor the health of the system by means of rich control dashboards.

Since 2018, the new term *observability* has been circulating in blogs and talks of the software engineering community and became trending topic among the companies providing distributed systems monitoring solutions. Observability has been considered from then on as a natural evolution of monitorability and has been associated with a practice that goes beyond the mere collection of telemetry data of a running application to estimate its performance. Observability should in fact guarantee a more granular real-time picture of the performance of the system under observation and of the possible problems arising during execution. Although observability may seem to be a synonym of monitorability, the community has always tried to keep the two concepts distinct from each other since the beginning. In [10] it is clearly highlighted how the term observability denotes a property of a system while monitorability is the act of observing (the performance or health of) a system over time [11]. Observability is therefore a measure of how much and well the *internal state of a system* can be *inferred* through a monitoring infrastructure [11]. The more observable a system is, the more relevant and useful the information extracted from monitoring is. For this reason, observability gained interest because it gives system administrators a greater control over the system as it enables a knowledgeable approach to monitoring.

Although the aforementioned sources give a better insight into the concept of observability, mainly by describing which data are typically considered by the infrastructure that realizes observations (metrics, traces and logs - see Sec. 2), they fail to provide a more general point of view on the concept of observability. Observability is in fact presented in a way that makes it appear as a specific property of service-based distributed applications and has never been adapted to different contexts. In addition, they do not clarify what is meant by “inference” or “estimation” of the state of an application. For the above reasons, extending the analysis of observability to a more general domain where applications are not necessarily service-driven and distributed is entirely reasonable.

The correlation between properties (functional and non-functional) of an application and the architecture has been acknowledged by the software engineering community and over time has become an established principle. Considering a quality attribute to be a measurable or testable property of an application, “Whether a system will be able to exhibit its desired (or required) quality attributes is substantially determined by its architecture” [12]. Adopting the interpretation proposed by [12], it is therefore valid to state that an application is observable if its architecture has specific structures (in the sense of [12], a structure is the set of functional elements and relations between them that determine the architecture) that make it concrete.

Overview of the work

In view of the above, we report on our experience in designing for and quantifying observable software architectures. From a motivational perspective, we recognise the intrinsic difficulty of making modern software architectural designs consistent with such properties, which often reflect conflicting design choices and still under-explored trade-offs. For this reason, we intend to identify what is the essential architectural scheme that an application must implement in order to be observable. Before doing this exercise, our investigation first proposes a definition of observability that is as general as possible and that is not directly ascribed to the world of self-adaptive and distributed applications. Our main outcome is that designing and catering for observability require specific and granular architectural solutions

such as Event sourcing [13, 14, 15], that can be concisely exemplified as *"Capture all changes to an application state as a sequence of events"* [1]. Event sourcing enables the continuous, incremental, and iterative improvement of the service functionalities in a DevOps fashion [16] by making service operations and their state explicit, in the form of analysable event traces [17]. At the same time, however, designing for observability and adopting Event sourcing are hard at best and represent, to date, a trial-and-error endeavor, with little to no reference approaches, tools, and techniques for designers to practically exploit as well as a myriad of possible problems [14, 15]. In fact, Event sourcing has always been described in general terms and it has not been accompanied by a formal definition.

In seeking to shed light over the aforementioned design process and its outputs, we enacted a mixed-methods study recapped in Fig. 1. Specifically, the figure uses an input-output flow-graph to highlight how we matched our

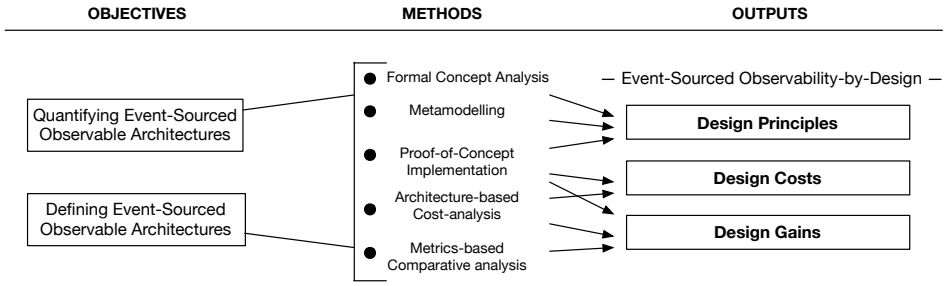


FIGURE 1 Event-Sourced, Observable Software Architectures: study overview.

general research objectives, as previously introduced, with research methods (middle-column in the flow-graph) and practical research outputs. Figure 1 highlights that, on the one hand, our general research goals are to address the problem of designing for software architectures which are—at the same time—event-sourced and observable. On the other hand, we interpret the act of software design as the process of making software architecture decisions which are as informed as possible, therefore requiring a *quantification* of the effects behind deciding for event sourcing as well as high-observability. At the same time, we interpret software architecture decision-making as the act of adhering—more or less strictly—to sound design principles and practices prescribed by any given technology, in our specific instance, event sourcing and high-observability software architectures.

Main results

From a methodological perspective, we first analyse—with a method tailored from formal concept analysis [18]—available definitions of event sourcing and high-observability software architectures to attain a formalization of the concept. Subsequently, we instantiate a practical and general overview of the aforementioned definition using the metamodelling and conceptual modelling procedure defined by Schewe et al. [19]. Finally, we operate a feasibility study via a proof-of-concept development exercise [20]. In particular, (1) we provide our formal definition of *monitorability* and *observability* from an architectural perspective in Sec. 3. This definition draws from previous work where we provided an outline of how observability can be made concrete in a classical architectural pattern like Model-View-Controller (MVC) [21], by showing a pattern refactoring that provides *observability-by-construction*. Subsequently, we introduce observability by design through Event sourcing and carve our approach to observability by applying the insights collected in Sec. 5. We investigate the abstractions necessary for making an architecture observable, hence distilling Event sourcing design principles. To this end, we introduce a metamodel that includes the core entities

[1] `https://martinfowler.com/eaaDev/EventSourcing.html`

essentials to build observable architectures. Also, we select suitable design patterns to materialize the introduced abstractions into tangible software components that allowed us to refactor MVC. (2) Finally, we study the cost of implementing observability in Sec. 6. To quantify the results of our experience, we present a first step towards a complete evaluation of the benefits and drawbacks of observability and Event sourcing at the architecture level. We offer an impact evaluation of observable architectures through: i) an experimental comparison study through a proof-of-concept implementation of observability adopting a Service-Oriented Architectures (SOA) architectural style [22] and contrasting the results with a related work [23] based on the Client-Server architectural style; ii) an architecture-based cost analysis featuring Function-Point counting; iii) a comparative analysis with metrics and measurements obtained by analysing our code through the CodeMR[2] software quality measurement platform. Our results indicate that observability is costly to design for, adding an increased effort of at least a factor >0.5 in at least 50% of the COSMIC function types [24] and yields an additional potential maintenance cost which remains currently unknown. Introducing Event sourcing in a non-observable architecture highlights some technical problems that the developer has to face in order to incorporate it, hence making the application observable. Firstly, since our definition of observability implies the presence of a mechanism that captures changes in the state of an application component upon the occurrence of specific events involving it, the developer must first address a conceptual problem, i.e. identify the relevant events that induce a change in the state of the application and, at the same time, the portion of the state of the application that undergoes the change and therefore needs to be tracked. Secondly, the developer is faced with a purely engineering problem that takes the form of implementing a series of functionalities to create a mechanism for tracking state changes. Event sourcing is in fact more an approach than a pattern; indeed, there is no a clear and precise publicly available model explaining the necessary abstractions to implement an Event sourced application (details are provided in Sec. 2). Hence, making observability concrete requires the design and the implementation of a number of abstractions that must be integrated into the application architecture and not just trivially superimposed. Our experience, which we propose in Sec. 6, shows that observability results in an architectural property of an observable application because the realisation of observability requires the development of specific functionalities and functional relationships between components that the unobservable version of the same application does not have. At the same time, however, observability makes architectures *explainable*, and should become a first-class citizen, especially in systems which demand increasingly the need to explain and expose their operations to the outside world. Altogether, our results encourage both practitioners and researchers to look further into the notion of architecture observability for automating systems maintainability, and other relevant Quality-of-Service [25] architecture properties in general.

Structure of the paper

Section 2 presents some related works. Section 3 introduces the definition of observability, while Sec. 4 describes the refactoring of a MVC pattern into an observable architectural pattern. Subsequently, Sec. 5 presents our research roadmap, the case study we considered while developing a metamodel of the essential entities that are needed to realize observable architectures. Section 6 illustrates a first evaluation of introducing observability by design. Section 6.4 presents the results we obtained, some discussion about our research solution and discusses its threats to validity, while Sec. 7 concludes our paper.

[2] `https://www.codemr.co.uk/downloads/`

2 | RELATED WORKS AND OBSERVABILITY: WHY DO WE NEED FOR A RIGOROUS DEFINITION?

2.1 | Observability and Monitorability: state of the art and challenges

The state of the art still offers a limited exploration of software architecture observability both from a speculative and experimental perspective, being observability a quite recent notion in the software engineering community. On the other hand, several recent works have touched upon cloud architecture complexity and monitorability in terms of microservices governance and management. Most prominently Toffetti et al. [26] offer experimental results on managing large-scale microservices solutions while Galletta et al. [27] offer a management solution but in the very specific domain of oceanographic Big Data management and analysis. Furthermore, Tamburri et al. [23] offer an operationalisation of the concept of *Observability* and offer a refactoring exercise that shows how software maintainability improves and at which costs. Further research is offered in this work, which extends [23] with a formal viewpoint of observability and monitorability and the evaluation of the impact of implementing observability in practice.

In Shahin et al. [28], the definitions of *loggability* and *monitorability* are offered: the former refers to the process of recording a time series of events of a system at runtime (as in [16]), whereas the latter consists of a process of checking the state of a system. *Observability*, however, is not mentioned and lacks of a formal definition that can precisely shape its meaning and distinguish it from the two mentioned properties. Even [28] does not offer a formal definition of monitorability, which is generally meant as a procedure that allows for checking system's state. Conversely, our definition of monitorability (see Sec. 3) is formally stated, and fits perfectly that of [28].

Some recently published articles attempt to better frame the actual definition of observability, mainly focusing on cloud and distributed service-based applications. All these sources seem to converge on a definition of monitoring (called Software observability in [29]), consisting in the collection and analysis of system data related to the performance of the application over time through an appropriate and specific application monitoring system, however not all of them clarify exactly what observability is. While [30] conceives observability as "a tooling or a technical solution that allows teams to debug actively their system" and puts the emphasis on a more managerial aspect of software development (thus conceiving observability as a DevOps tool a team must use to "explore properties and patterns not defined in advance"), [10, 11, 31, 29] agree that observability is a property inherited from engineering and control theory. A closer look at this definition is therefore worth of mention. The definition of observability by Kalman [32] is peculiar to the theory of dynamical systems and defined through specific properties of differential equations modeling the system. In such a context, the dynamics of a system is described with time-dependent variables over the Reals that represent internal system states $\overline{x}(t)$, system inputs $\overline{u}(t)$ and system outputs $\overline{y}(t)$ (the overline represents a vector of variables). A system is *observable* if it can be described by means of a system of differential equations in which it is not possible to partition the state variables into two groups $\overline{x}^1$ and $\overline{x}^2$, such that the second group $\overline{x}^2$ affects neither the first group $\overline{x}^1$ nor the outputs of the system $\overline{y}$. In other words, a system is observable if all the state variables contribute to the definition of the output values. The notion of observability has been adapted to the context of software, and turned into the "measure of how well internal states of a system can be inferred from the knowledge of its external outputs". Therefore, "a system is *observable* if the current state can be estimated by only using information from outputs, namely sensor data". In the context of distributed service-based applications, observability helps developers identify and resolve performance issues as it allows for "exposing the state of an application" while it is running. To achieve observability, the community agrees on the use of tools to collect the following data at runtime: *metrics*, *traces*, and *logs* (they are called "Pillars of observability"). Roughly, a *log* is a textual description of an event that happened at a particular time; it includes a timestamp and a payload that provides context. A *metric* is a numeric value measured

Pattern	Implemented functionality
Log aggregation	Centralized logging service that aggregates logs from each service instance. Searching and log analysis are available.
Application metrics	A service which collects statistics about operations and provides reporting and alerting.
Audit logging	A functionality that records user activity in a database.
Distributed tracing	Instrumentation with code that allows services to track every external request with an unique identifier (i.e. to realize traces).
Exception tracking	Centralized exception tracking service that aggregates and tracks exceptions and notifies developers
Health check API	Endpoint (e.g. HTTP /health) that returns the health of the service.

TABLE 1 Available observability patterns in [33].

over an interval of time and refers to a quantitative aspect of the application, commonly related to performance indicators and a *trace* represents the journey of a request through a distributed system and the functions implemented therein. In [33] several patterns (summarily described in Table 1) are presented as solutions for implementing (facets of) observability, i.e. that enable the designer to implement the infrastructure that collects metrics, traces and logs. For each pattern, [33] shows a brief description of the potential problems that its implementation in practice may induce. Even if the description is general and does not provide specific details, the problems one faces in implementing these patterns can be bundled in two main ones: on the one hand, there is the need for significant infrastructure to store traces and logs and, on the other, the intertwining between the pattern code with the business logic of the application. In the next sections, we try to generalize the notion of observability without overlooking the current literature and attempt to undertake an exercise that allows us to give concrete meaning, through Event sourcing, to what the articles cited above define as inference/estimation of the state of an application.

From an industrial perspective, prominent technical reports from Dynatrace and Twitter highlight the need for specific and organizational-tailored approaches to achieving application observability including an elaborate strategy which features: (a) organizational elements and clearly defined roles both at Dev and Ops stage; (b) appropriate tooling for monitoring and closing-the-loop (i.e., feeding runtime information back into design and development-level refactoring activities); (c) synergistic architectures designed to accommodate both (a) and (b). More specifically, Twitter highlights the necessity to address their scalable systems observability with a granularity at least at the level of microservice dependencies[3] which is key to maintaining Twitter-base and connected services operational. Conversely, Dynatrace reports that so-called *unknown unknowns* require multi-granular observability features and coherent organisational strategy around the architecture-level constructs. Quoting from Dynatrace: “[addressing] the kind of unique glitches that have never occurred in the past and cannot be discovered via dashboards [lead to] the growing pains that the concept of traditional observability attempts to tackle".

In [34] the author introduces the notion of *visibility* in the context of network-based applications in which “communication between components is restricted to message passing [35]”, or to an equivalent of message passing selected at runtime based on efficiency criteria and location of involved the components [36]. “Visibility, in this case, refers to the ability of a component to monitor or mediate the interaction between two other components. Visibility can enable improved performance [...], scalability [...], reliability [...], and security [...].” Although the concept of visibility seems to be delineated and distinguished through two specific functionalities, i.e., monitoring and mediation, it does not provide these with a precise definition that makes the concept of visibility clearly defined.

[3] https://blog.twitter.com/engineering/en_us/a/2013/observability-at-twitter.html

2.2 | Model-View-Controller and Principal Variants

In Sec. 4 we describe the refactoring of a MVC pattern into an observable architectural pattern. MVC has been the reference architectural pattern for most web-based applications for decades and, given its prominent use, it has seen several variants. The first variant is known as Model-View-ViewModel (MVVM), that was developed in Microsoft for building event-driven user interfaces [37], on top of a data-binding technology[4]. MVVM defines an abstraction, called ViewModel, besides the classical Model and View abstractions from classical MVC, and rule out the role of Controller, being it replaced by *Binder* that automates communication between the view and bound properties in the ViewModel. In essence, Binder is not a proper component, as it is commonly implemented by means of markup languages, rather it is an underlying framework which exploits data binding functions (e.g., Windows Presentation Foundation in its first definition). The purpose of ViewModel is to provide a view of the data in the Model (i.e., a converter) that View can easily manage and present. It decouples View and Model with a local state, used by the Views to render a proper UI, and a set of commands used by the Views to signal the occurrence of a user interaction. The execution of the commands causes the ViewModel to trigger the execution of part of the business logic, which might imply a new application's state. When Model change, ViewModel and View are notified and updated accordingly.

The Model-View-Presenter (MVP) is another variant of MVC that aims to generalise MVC for building user interface architectures in Java and C++ [38]. MVP defines the following abstractions: *Model*, *Command*, *Selection*, *View*, *Interactor* and *Presenter*. Model is like the one of MVC, but it mainly stores the state of the application. The application behavior is outsourced to the Command abstraction, which offers an interface for updating the Model. Selection represents elements of the application's state that Commands can change. The purpose of View is the same as MVC, but in MVP, it is also capable of capturing user interactions through Interactors (in MVC user interaction is captured by the Controller). Presenter is the coordinator of all the previous abstractions.

To emphasise the difference between our solutions and the known patterns, we point out a list of characteristics of the patterns that we consider as potential limitations to a correct architectural approach to observability. MVC does not define distinct abstractions representing the application's state and the procedures manipulating it. Model actually groups both together, limiting decoupling. Moreover, in MVC there is no abstraction to represent external events as the state transition is enabled by generic messages, exchanged by Controller and Model. Model-View-View-Model (MVVM) offers similar features as MVC. Also, since it relies on data-binding frameworks for the event processing, it requires customization if no underlying frameworks are available. Model-View-Presenter (MVP) does not provide an abstraction to decouple the notion of event and actions that should be taken to handle the event, rather it provides an abstraction –Interactor– to signal a user interaction to Presenter. Finally, all the MVC-like solutions do not isolate a specific component for tracking the evolution of the application's state along the executions.

2.3 | Event Sourcing Explained

Although the software engineering community lacks rigorous definitions of observability, possibly accompanied by clear design criteria and defined through precise architectural structures, the industry has been long referring to an architectural pattern whose function is suitable for implementing the notion of observability that we propose in the next section. This architectural pattern is called Event sourcing and has been known in the software development community for a long time, as shown, for instance, by several blog posts and articles available on the web [39, 40, 41].

[4] Data binding is the process that establishes a connection between the app UI and the data it displays. If the binding has the correct settings and the data provides the proper notifications, when the data changes its value, the elements that are bound to the data reflect changes automatically (https://docs.microsoft.com/en-us/dotnet/desktop/wpf/data/data-binding-overview)

It must be pointed out that Event sourcing is not an observability pattern and the goal of this work is not to reconsider its nature; rather, the goal is to use its feature to implement observability. Event sourcing is, in fact, an approach to managing the temporal evolution of the application data (or, more in general, of the application state), and can be considered as a data management architectural pattern. It may be adopted in applications where the data is isolated in a self-contained entity, such as a dedicated storage object or database, which manages the data by applying operations obtained from sequences of events to be handled by the application logic. Event sourcing is, in fact, commonly coupled with Command Query Responsibility Segregation pattern to separate read and update operations for a data store in highly performant and scalable applications [42, 43, 44]. Citing Fowler, "event Sourcing ensures that all changes to application state are stored as a sequence of events" [39].

Developers envision Event sourcing as a solution to some limitations of traditional implementations which change the current state of the data as soon as operations are received (e.g., a CRUD operation), possibly by adopting transaction mechanisms that lock the data. According to [40], there are possible drawbacks if this approach is used blindly. Performing operations directly on the database "can slow down performance and responsiveness, and limit scalability, due to the processing overhead it requires." Also, "in a collaborative domain with many concurrent users, data update conflicts are more likely because the update operations take place on a single item of data." Finally, no history of the performed actions can be available, unless auditing mechanisms are specifically implemented.

Even if no common architectural pattern is publicly available, all the sources seem to agree on the presence of a component called Event Store that performs the persistence of events driving data changes. Event Store implements publish/subscribe features that allow other application components to get notified about required data modifications. Event persistency enables replaying events in the store in order to create views of data, to rebuild entirely the current state of data in case of system failures, or to re-compute the state by reversing the effect of past events that later in the execution were discovered to be wrong [39]. In addition, since events are simple objects that describe some action (together with context information), and do not alter immediately the data upon their arrival, concurrent data updates can be prevented, as handling event occurs at the right time and updates are not directly applied to data. As a result, the presentation and the logic tiers can improve performance and scalability, and the obtained decoupling of the operations on the data from the event management provides flexibility and extensibility. [40]

Event sourcing is not a panacea for all the ills and inherently brings some limitations. The delay between the addition of an event to Event store and the time when consumers handle the event can lead to inconsistency between the current version of the data and the version obtained by applying the events that have arrived in the meantime. When applications are complex and maintain several copies of data, or views thereof, in a multi-threaded or concurrent architecture, the logic of data management must still be able to deal with inconsistencies that result from eventual consistency and from the fact that the use of transactions is discouraged. In fact, Event store can be used by several distinct components to create internal views of the data, and strong consistency, when implemented, is often a hurdle to efficiency. Moreover, the consistency of Event store per se is paramount. Multi-threaded application should manage it through suitable mechanisms (e.g., timestamps), and the reversing of the effect of an event requires suitable mechanisms (e.g., compensating events), because Event store should never be updated retrospectively as the event history cannot be altered. The interested reader can find more details in [40, 39].

In summary, a mapping between the most essential related work and our contributions and characteristics of our study are reported in Tab. 2, which relates all state-of–the-art approaches (columns) across the seven essential dimensions (rows) that the same state of the art has focused upon up to this point, namely, scale, domain-specificity, refactorability, definition operationability, principled design, low-code and application to real-life industrial case-studies. The table compares major state of the art contributions in the last few years along the lines of the six essential characteristics either introduced by or claimed as open limitations within each of the aforementioned works. Overall,

Features	Related Work						Our Work
	[26]	[27]	[23]	[28]	[34]	[32]	
Analysis at Scale	✓	×	×	×	×	×	×
Multi-Domain	×	×	×	×	✓	✓	✓
Refactoring Focus	×	×	✓	×	×	×	✓
Defining	×	✓	×	✓	✓	✓	✓
Design Principles	×	✓	✓	✓	×	✓	✓
Model-Driven/Low-Code	✓	×	×	×	×	×	✓
Real Case-Study	×	✓	×	✓	✓	×	✓

TABLE 2 State of the art, an overview and comparative analysis.

our synthesis reflects the need for more precise and rigorous definitions and tool-support as well as design constructs to approach a more varied and comprehensive notion of observability are required and both the state of the art and practice cannot currently deliver properly on such required constructs. We argue that designing for observability is a required stepping stone as part of any DevOps process supporting a sufficiently complex architecture that would justify the extra architecture rigour. Our contributions start from such a definition and work to provide solid and tested-true principles that can aid in refactoring architectures towards increased observability. Also, our results aid in providing architecture-level constructs that help software architects and other DevOps stakeholders address such additional architecture rigour.

3 | SOFTWARE ARCHITECTURE OBSERVABILITY

The lack of a precise definition of observability and monitorability of software has been carrying on the discussion in the community about which aspects actually distinguish a monitorable application from an observable one. Several are the attempts towards the provision of a precise understanding of observability and monitorability, that have been mainly shown by practitioners through a mottled amount of blog posts and articles[5]. We provide a more theoretical interpretation of observability and monitorability by reinterpreting the common understanding of these two aspects through an abstract representation of the behavior of (software) systems.

3.1 | Definition

Consider an implementation a, and let S be the state space determined by all the possible configurations of the memory that are determined by a running instance of a. The definition of S is not limited to the "static" variables rather it is general, and an element of S can be seen as the memory footprint of a at a given time throughout the execution (including CPU registers). Let I be the set of instructions in a, and let $\xrightarrow{i} \in S \times S \times I$ be the transition relation determined by the implementation a, which is such that $s \xrightarrow{i} s'$ holds when the execution of instruction i from state s modifies the application state into s' (let $\rightarrow^*$ be the transitive closure of $\rightarrow$). We say that a sequence of states $s_0, \ldots, s_{n-1}$ in S is an *execution* of a if the states are pairwise related through $\xrightarrow{i}$, i.e., that are such that $s_i \xrightarrow{i} s_{i+1}$ for every $0 \leq i < n-1$, and s_0 is an initial state. Let E be the set of all the executions of a. Consider, for instance, a code

[5] https://landing.google.com/sre/book/index.html

```
class X{
  t_A att;

  void f(int t, int q){
    int h = t;
    while (h > 0){
      att.setter(h);   // this call has a side effect on att
      h = h-q;
    }
    if (att.r() == 0)
      att.getter();   // this call has no side effect on att
    return;
  }
}
```

FIGURE 2 observable code snippets; an example.

including two integer variables called x and y and that a state of a running instance of the application is defined by the tuple $(x = 2, y = 5)$. Assume that variable x is incremented by means of instruction $x = x + 1$. Hence, we can write the state transition as $(x = 2, y = 5) \xrightarrow{x=x+1} (x = 3, y = 5)$.

In the following the definition of monitorable and observable application is provided. Let U be a subset of E. Intuitively, the subset U is the set of executions in E satisfying specific (non-functional) properties such as, for instance, CPU intensive executions because of specific input data or particularly unfavourable execution environment. In this scope, the following definition applies.

Definition. We define an application a as *monitorable* if there exists an implemented function f_{m} from E to the set of natural numbers $\mathbb{N}$ that indicates if an execution of a running instance of a belongs to U. Conversely, we say that application a is *observable* if, given an execution $s_0, \ldots, s_{n-1}$, there is a function f_{o} from $\mathbb{N}$ to S that is equal to s_i if i is a position of the execution.

Stemming from the definition, the goal of monitorability and observability can be clearly plotted. While the former aims at providing functionalities that enable the identification and classification of the executions of an application, the latter only supplies a detailed view of the application state at a given position throughout an execution. In other words, an application is monitorable if there is an implemented functionality (i.e., function f_m) which can evaluate at runtime the value of some software-related metrics that the current execution of the application is determining. Among many, we can mention for instance, the CPU usage, the memory footprint, the number of incoming requests per seconds, the number of delivered outcomes per seconds, etc. The definition of U is general in the previous definition, whereas the common understanding is more specific. Set U, in fact, usually includes the executions that might generate future failures, or indicate unhealthy configuration of the runtime environment. It becomes clear when one or more metrics are selected and their characterization is given. For instance, set U can be the set of all the executions of the monitored application (or component, function, service, etc.) such that the service output rate is less than 1 million processed requests per seconds. For this reason, the occurrence of an execution in U should be recognized by monitoring

components, and properly managed at runtime, to avoid unpredictable evolutions or performance drop. We can assume that the natural value returned by function f_{m} is associated with a specific "explanation" that characterizes the criticality of every execution in U. By using this information, specific components can react proactively to prevent failures, or to improve the health of the application, as soon as the monitoring components detect a problem. This interpretation is fully compatible with the definition in [28], and the common understanding of monitorability.

Observability has a different goal, that is complementary to the one of monitoring. Observability, in fact, is not a substitute for monitoring, nor does it obviate monitoring. An observable application has in fact specific components that make the reconstruction of the state of the application possible, whatever position throughout an execution is selected. The information obtained from function f_{o} can complete the one provided by the monitoring components implementing f_{m}, hence allowing for a better understanding of the causes that negatively affect a running application. In minimalist terms, an application is observable when any change in its state is recorded in a log, along with the event that caused the change.

Our definition of application is general, and it does not tie the term to one instance of a program. Indeed, by enlarging the scope of term application, we can include, for instance, operating systems or virtual machines that take care of the execution of several distinct applications, but also services or distributed stateful functionalities. Therefore, when observability has a larger scope than a single application instance, the information that can be collected by means of f_{o} can be very rich, and provide insights into the behavior of a specific application along with a rich context, determined by a complex environment that might include several concurrent applications.

From a practical perspective, f_{m} and f_{o} can hardly be implemented when the size of the implementation a is realistic. Procedure for monitoring or observability cannot watch over everything, and their implementation can become so complex and pervasive that use and maintenance can become complicated and costly. Indeed, the burden of tracking too many details of state of the running application can have a drastic impact on the performance, and can easily lead to a misuse of resources. Monitorability and observability in practice are attained by considering a proper projection of the application states. A possible projection, that can be implemented at the application level (i.e., that does not require low level calls to the SO), can be defined as follows. Let V be the set of all variable identifiers that occur in a, and let $\bar{V} \subseteq V$ be a subset. Given a state $s \in S$, the projection of s on $\bar{V}$, indicated as $\pi_{\bar{V}}(s)$, is the portion of state s only related to the variables in $\bar{V}$ (we can assume that different values in s that are bound to the same variable identifier, e.g., because they reside in distinct activation records, are distinguished by a suitable indexing).

We exemplify our interpretation of observability by interpreting the term "application" as "class". Consider the class `X`, whose implementation is shown in Fig. 2, which includes an attribute `att` and a method `f`. Intuitively, class `X` is observable if it is possible to *observe*, i.e. record, the effect of the execution of `f` on the value of attribute `att`, for a specific instance of `X`. Since only the instruction at line 7 can change the value of `att`, if the value of the attribute `att` is recorded after every call of method `setter`, the effect of `f` is fully observable.

3.2 | Implementing observability through Event sourcing

Event sourcing represents an approach to observability; if suitably employed, it allows to achieve observability by design. In fact, the function f_o can be effectively implemented by adopting Event sourcing as an archetype that allows developers to identify the main entities that together provide observability. To obtain observability, an application can implement an event store component which records all the relevant events that entail a state change. Hence, the presence of a log of events and a state s_i makes it possible to compute state s_j by replaying the events from i. The granularity of the observations that determine the definition of the events is application dependent. It is up to the developer to identify the proper set of events that should be observed, together with their nature and composition.

The nature of an event, in fact, can be a single statement or even a sequence of function calls; and, transversely, an event can be determined by a non trivial composition of occurrences such as, for instance, two function calls f and g, one following the other within a given context (e.g., f returns some data). Therefore, implementing observability via Event sourcing is far more than a log that records events driving a state change, but it requires the recognition of:

- the part of the application (e.g., the components) which determines the projection of the application state to be observed, and that change their internal state upon the occurrence of an event,
- the events, their nature and composition; and the context that they bring over, i.e., the information they use to realize the state change,
- the entity that enforces a state change in those that are affected by the occurrence of an event,
- the entities that provide the logic to handle an event, and finally,
- the entity that realizes the information persistence, which renders the information realizing the projection of the application state and the events that drive the state changes into a machine readable log, and that allows for the reconstruction of the application's state.

Our definition of observability is general enough and the patterns presented in Sec. 2 can all be seen as an instance of our definition. For every observability pattern presented in Sec. 2 and described in [33], we can characterize what constitutes the concept of state, which are the events that reasonably might be those inducing a state change to be recorded and which is the source/entity that can cause the events. The latter two entities in the list of elements that designers should identify to implement observability via Event sourcing (presented above), i.e., the one providing the logic to handle an event and the one realizing information persistence, are not shown because their functionality is general and does not depend on the specific pattern's functionality. In other words, we safely assume that all patterns may be considered to implement a logger and some control logic to deal with events and to enforce a state change. The result of this analysis is shown in Table 3. The column "State" describes which notion of state is being considered in a pattern and that should be observed in order to realize the pattern's functionality. The column "Events" suggests possible relevant events that determine changes of the state. Finally, the column "Source" indicates viable entities that might trigger the events relevant to the functionality realized by the pattern.

In Sec. 5.3, we show a metamodel which captures all the relevant entities that can be used to implement observability by an Event sourcing -based approach, and how the abstractions can be mapped to software components. Since there are no precise abstract models capturing Event sourcing we implement the mapping by using a well-know design pattern called Command.

4 | OBSERVABILITY AS AN ARCHITECTURE QUALITY: THE MVC CASE

The definition of observable application that we set forth in Sec. 3 is abstract, and cannot be immediately transferred to a working implementation. Our definition does not actually identify the functional requirements that an observable application should put into effect, nor establishes how the functionalities realizing observability should be translated into an architectural pattern.

In general terms, observability can be considered as a software quality that materializes through an integrated implementation of function f_o. As already discussed in the previous section, our definition is general, and therefore,

Pattern	Observed entity	Observed entity's State	Events	Source
Log aggregation	The application's loggers	The information that characterizes an error or message to be recorded and, possibly, a projection of the application's state (e.g., division by zero because variable x is null)	An internal or external (w.r.t the application) contingency that causes an error or message to be logged and the context that can explain the contingency. Events may not directly generate an error/message but contribute to its occurrence (other events that precede the one that triggers the error but are somehow related to it).	Every entity of the application that requires the execution of the logger functionality
Application metrics	The monitoring system	The information that characterizes the metrics to be recorded and, possibly, a projection of the application's state (e.g., CPU usage, allocated physical memory, etc.)	The activation of a timed recurrent system's health checking procedure or any generic functionality requiring system resources (e.g., a timed function call which monitors the system health, the occurrence of a process creation, object allocation, etc.) and the context that can explain the contingency	The ticker component calling the system's health checking procedure, or the system's health checking procedure itself, and any software entity that requires system resources
Audit logging	User activity database	The database instance	Every user action	The user interface or any component that processes input from the user and its requests subsequent to their acquisition
Distributed tracing	The service request trackers and/or the centralized component which records the requests' data	The information that characterizes the service requests to be recorded and, possibly, a projection of the application's state (that might be collected by other application's components)	An occurrence of any service request that any generic component/service in the application may have to process	Any service/component in the application
Exception tracking	The local/centralized exception loggers	Similar to Log Aggregation pattern	Similar to Log Aggregation pattern	Similar to Log Aggregation pattern

TABLE 3 Framing observability patterns [33] into our approach.

it is not compulsorily restricted to an application or a part of it. Hence, it is perfectly valid to observe an object through a suitable projection of its variables, but also through an entire application. In any case, the integration among the observed application entities and the part of the application that realises f_o expects all the involved entities to implement specific functional interfaces, which directly influence their implementation. Since styles and software architectures per se determine the roles of the implemented entities and their interactions, observability can then shift to the architectural layout of an application and become an architecture quality. Therefore, we say that an architectural pattern offers observability (or simply, it is observable) when it defines explicit abstractions that fulfill the following Functional Requirements (FR):

FR1- operating with a machine-readable representation of a set of variable values, in which the system is observed to be in any instant during operation [45], and with a machine-readable representation of the events (data input or internal operations) that have an effect over that variables;

FR2- rendering the architecture state into a machine-readable representation to the outside environment.

Given the need for integration of f_o with the observed entities of an application, and its effect on the implementation, any implementation of f_o can also be seen as a software "aspect", being it a feature that lies transversely to both the core and non-core parts of an application, and that is not directly related to the application's primary functions (i.e. a *cross-cutting concern* [46]). Besides Aspect-oriented design/programming (AOD/P), which is considered as a possible approach to architectural modularity, one can take advantage of off-the-shelf platforms such as Microsoft Azure [47] or Eventuate [33]. While the former requires no specific introduction, the latter is a platform for developing "transactional business applications that use the microservice architecture" which "provides an event-driven model that is based on Event sourcing and CQRS". The platform is based on a SQL database for event persistency and Kafka [48] as the publish/subscribe mechanism (an example application implemented with the open source platform can be found in [49]).

Despite the variety of technological solutions and architectural approaches, the goal we want to pursue in this work is beyond the mere adoption of a design/programming approach, which we simply consider as a means for developing a software solution. Our goal is more ambitious, as we want to understand how the abstract notion of observability can be made concrete, which are the essential software entities and relations that designers should identify to obtain observability, and the cost they come with, regardless of the targeted design/programming approach. To this end, we transfer the prescriptions identified in the next sections to an effectively modular software architecture of a realistic application, and we derive a metamodel in Sec. 5.3, by adopting an inductive reasoning, which characterises all the essential entities and relations required to render observability into software. In essence, adopting a different design paradigm is clearly possible, provided the meta-model elements are mapped to actual software elements that are specific of the selected approach.

To implement the feasibility study via a proof-of-concept development exercise in Sec. 5.2, we ultimately opt for existing architectural styles [50], and we adopt the client-server and component-based styles. Both are very well-known and popular. In addition, the latter is general enough to be transversal to problem-specific approaches. It is sufficiently expressive to allow the development of an effectively modular software solution, and it is also common in different programming languages and paradigms. Using a component-based style allows us to render the meta-model entities into actual components and exploit OO design-patterns.

Finally, after showing the observability meta-model, we evaluate the overall architectural cost of a realistic application which is induced by the entities and relations in the meta-model that supply for observability.

oMVC	MVC	MVVM	MVP
Store	Controller	Binder / View-Model	Presenter
State	Model	Model	Model
Resolver	Controller	Binder	Presenter
State Policy	Model	Binder	Command / Selection
Side Policy	-	-	Command / Selection
Action	-	-	-
View/Normal	View/Controller	View / ViewModel	View / Interactor
Logger	-	-	-

TABLE 4 Mapping of oMVC abstractions.

4.1 | Motivations for Model-View Controller

To offer a authentic example of observability as an architecture quality attribute, we select the most recurrent architectural pattern known to date—the classic Model-View-Controller pattern [21]—and we introduce its refactoring which provides for observability by-construction [23].

The selection of MVC rests not only on its nature of most used design pattern both in practice and research [51, 52] but, from a technical perspective, on the following two traits: (1) MVC is the de-facto standard when implementing reactive web applications and portals, and therefore it reflects the most modern cloud-native digital application architectures alike; (2) MVC is sufficiently high-level and often referred to as an architecture-level design pattern, since its design rules may generalise to many subsequent design choices (e.g., the adoption of a client-server approach for specific areas of the MVC-structured web portal).

4.2 | Observable MVC Explained

Observability actually reflects an ordered set of data transformations [53] affecting the state of a running application. Considering the previous arguments and stemming from the definition of Sec. 3, we need to modify MVC to include the functional elements that enable Event sourcing and that we identified in Sec. 3.2. In the following, for each of them, we show which components in MVC achieve the specified functionality, and how they provide it, and which new functional abstractions not present in MVC are introduced to build an Event sourcing -based architecture. We point out first that some functionalities provided by the MVC components are rearranged in oMVC and mainly split into distinct functionalities realized by oMVC components. Table 4 shows the relationship among oMVC and the patterns MVC, MVVM and MVP. Every row indicates the component in oMVC which realizes part of/the same functionality implemented by components in MVC, MVVM and MVP. The following abstractions materialize the functional elements described in Sec. 3.2.

Modeling the evolving application's state to be observed

The MVC Model provides both the functions (changes to the application's state following a request from the Controller component). In oMVC, the *State* abstraction represents the observed state of the application, i.e., the set of pieces

of information that constitute the projection $\pi_{\tilde{V}}$ defined in Sec. 3. The latter function is entrusted in oMVC to a functional abstraction separate from State and dedicated only to the state update function (see *state Policy* later).

Modeling the events causing a state change and their instantiation

MVC has no abstractions that model events but one, the View, to capture external events (e.g., user's requests). In oMVC, the execution of a business logic paired with entities implementing the *Action* abstraction entails the evolution of the application state. Relevant (internal or external) events are intercepted by the *View/Normal* components, which make them available to the rest of the architecture as Action. View/Normal components take over the MVC View, adding a more refined interpretation of this component. oMVC View components intercept external events and produce a representation of the application's state (via the State) to be communicated only to the external world; while oMVC Normal ones take care of internal events and build a representation of the application's state that is intended only for other application's components. Similarly to MVC View, a View/Normal component might produce a new encoding of the application state whenever the effect of the execution of an Action alters the application state.

Handling events with proper logic

The MVC Controller provides the logic to handle events and controls the execution of the logic realizing the application's state change along with the more general coordination and supervision of the Model and View components. In oMVC however a finer functional partitioning is considered and the *Command* pattern [54]. is adopted to decouple the logic to handle events from the one enforcing the state change. In the Command pattern, the Command interface usually declares one method for executing a command and its implementations are instantiated to handle specific requests. The logic that defines the application's state transformations in oMVC, hence including the business logic of the application, is encapsulated into the *Policy* abstraction, which corresponds to Command interface. Every Policy is equipped with method *apply()* which computes a new application's state based on the Action to be handled. Policies are categorized into *state* Policies and *side* Policies. The former implements procedures that modify the application state, while the latter implements the logic of actions which read the state without manipulating it (e.g., querying some information from it). In the Command pattern, two distinct roles use the command objects: the Sender, which intercepts an event and triggers a computation to handle it by means of a command, and the Receiver, a component which implements the proper logic to deal with the event. Nonetheless, part of the logic can also be implemented by the command object itself, complementing, therefore, the part that can reside in the Receiver. In oMVC, Sender is impersonated by the component which triggers a state change (see Store later), and Receiver is realized by the classes constituting the Model. Besides policies, oMVC includes the *Resolver* abstraction, which represents the mapping between an Action and a (state and side) Policy that should be executed to manage the Action.

Enforcing of a state change upon the occurrence of an event

The MVC Controller provides this function, which is committed to the *Store* abstraction in oMVC. The Store component in fact enables the modification of the application's state by running the policies (state and/or side) that are identified by the Resolver component when an Action has to be handled. In addition, the Store component is responsible for notifying View components about the change, so all the views can provide a new data representation to the external world. Method *propagateAction()* in Store implements the logic of a policy that handles an occurred event.

Reconstructing the application's state

MVC is devoid of such a functionality. In oMVC the *Logger* abstraction handles data persistence and abides by Store instructions, that specify which data have to be recorded and how the application's state can be rebuilt. According

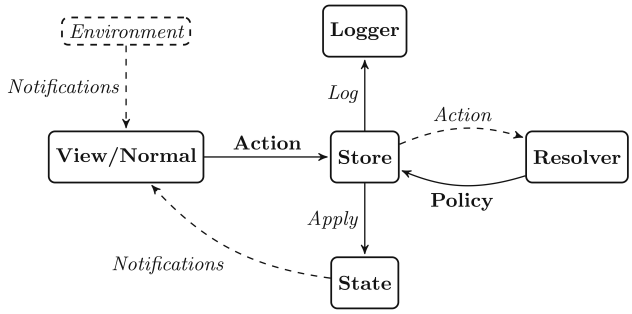


FIGURE 3 Data flow in oMVC. Boxes represent the main components of oMVC, whereas arrows indicate an information exchange. In particular, solid lines imply the presence of an instantiated object, such as an Action or a Policy; or a functional dependency that is realized between two components (e.g. Apply and Log). Dashed arrows represent generic dependencies that can be designed as application-dependent features.

to the definition of observability in Sec. 3, an application is observable if the reconstruction of an occurrence of the (projected) application state, manifested throughout the execution of the application, is allowed. To implement such a function, first all the state configurations that occur in an execution should be persisted in a log along with the Actions that transformed a state into its successor. Second, the reconstruction of an instance of the application's state should be performed using the recorded data in the log. The Logger component, which is activated by the Store when a State policy is issued, guarantees the persistence of the application state Every time a state Policy is applied on the state, the Store component can use the getter methods of State to collect the projection of the application's state that should be logged, before and after the execution of the Policy. The Logger component records the state change with tuples of the form (a, s_i, s_{i+1}) that include the Action descriptor a, and a representation of the application state s_i and s_{i+1} before and after the execution of the (State) Policy, is appended in the log. Action a and the pair of states s_i and s_{i+1} are both represented based on a suitable encoding. The application's state reconstruction in done by the Store component based on a sequence of Actions, that are replayed from a position in the log, that is inherently associated with a specific state. We observe that the definition of the log tuples, as well as the implementation of the function that reconstructs the state, are generic in our definition. Both, in fact, are implementation specific, and should be customized according to specific design needs. The State component can include, for instance, simple numerical entities but also complex data structures; whereas Actions might be basic transformation (e.g., an increment of an index) or more complex operations (e.g., the elimination of a node in a balanced tree). The application designer should, therefore, adopt the most suitable encoding for representing state variables and data transformations, and finally identify the best trade-off between the amount of information collected in the log, and the amount of computation committed to the procedure that restores the application state.

Figure 3 offers the data flow which ensues from our refactored—and observable—version of MVC (i.e., oMVC or "observable" MVC), introduced in [23]. The occurrence of an event outside the application, such as a request for a data change within the application, or an internal event, is captured by a View component and encoded via an Action object along with the request-specific data. The View notifies the Store component which then requests a Policy object from the Resolver component to process the request. The Policy object applies to the current state and calculates the new application state which overrides the current one through the action of the Store component and the collaboration of the State component. The application's state change is recorded using the Logger component at the behest of the

Store component.

5 | STUDY DESIGN AND EXECUTION

As clear from the aforementioned concepts and definitions, designing for, and implementing architectural observability comprises a considerable cost with still relatively unclear consequences over other key architectural characteristics, such as maintainability and more. A full elaboration of these consequences and an exploration of best practices and approaches to manage the architecture trade-offs connected to observability are yet to be uncovered. Consequently, in this manuscript, we aim to address the following research questions:

RQ0. Which are the abstractions needed for making an architecture observable?

RQ1. What costs are implied when designing for observability?

RQ2. What maintainability consequences does design-for-observability manifest?

In the scope of RQ0, we aim to define the core abstractions that are essential for implementing architectural observability. In the scope of RQ1, we aim at understanding whether there are costs connected to designing for observability and what those costs might be. Furthermore, in the scope of RQ2, we aim at understanding and measuring the architectural consequences—in terms of global architectural software metrics—that manifest in connection with a fully-observable software architecture.

5.1 | Research Methods

To attain our results, we exploit the work in [23] in which a standard exploratory prototyping approach was used. While in [23] the aim was on re-implementing a prototype conforming the Client-Server (CS) architectural style, in this study, we focus on a simple yet expressive industrial case, a service-based weather prediction and data analysis system (description in Sec 5.2). This is a distributed service-based application featuring a more elaborated Service-Oriented Architecture realizing its services through RESTful APIs. The evaluation has been carried out by considering two different implementations of the same case study; one is based on the standard MVC pattern, while the second builds natively in the oMVC pattern[6]. Our focus on this rather simplistic case is a specific research design choice. We intend to explore the impact and consequences of designing for observability in a very simple architecture prototype, albeit with several appealing and controllable architecture features, such as service-based style or number of components. We in fact assume that such a simplistic case offers a very optimistic estimation of observability consequences, which are bound to be higher in more complex systems.

To answer RQ0, we devise a metamodel that includes the core entities which are essential to build observable architectures. For the sake of clarity, the metamodel is presented by explaining its constituents through the new developed application, yet it reflects all the entities that are proper of oMVC, and in turn, those that implement MVC, MVVM and MVVP and that can be implemented by oMVC entities (see relation in Table 4).

To explore RQ1, we adopt a state-of-the-art Function-Point (FP) analysis [55, 56] approach known as COSMIC, stemming from an international standard for measuring software size, elaborated on standard ISO/IEC 19761:2011.

[6]The two implementations are available at `https://github.com/nicologhielmetti/weatherdatanalysis_oMVC`, `https://github.com/francescoalongi/weatherdatanalysis_MVC`

The count essentially focuses on data movements and transformations within a software architecture and across components, with four data movement types, Entry, Exit, Write and Read that typically require measurement. To further substantiate the FP counts connected to the COSMIC procedure, two observers were involved using the well-known Krippendorff$_\alpha$ approach [57]. The α score essentially measures a confidence interval score stemming from the agreement of values across two distinctly reported observations about the same event or phenomenon. In our case, the value measures the agreement between the two COSMIC-based FP measurements for the case under study. The value was calculated initially to be 0.89, hence $\alpha > .800$, which is a standard reference value for highly confident observations. Subsequently, the value was used to drive the agreement between the two analyses up towards total alignment[7].

Finally, to explore RQ2, we evaluate the comparative analysis with metrics and measurements from the CodeMR software quality measurement platform, a holistic software measurement platform, which offers a number of software architecture quality metrics, such as coupling and cohesion between modules [59, 60]. Subsequently, the measurements in question were analyzed comparatively with descriptive statistics to evaluate the variance between the two prototypes connected the observability of one of them.

5.2 | Observability Implemented: a Case-Study

The new case study concerns a service-based weather prediction and data analysis system, realized as a three-tier web application that is intended to collect and store weather data, emitted by remote stations, and supply functionalities for statistical analysis via the web. The user of the application should be allowed to: 1. *create a station* in the database; 2. *upload data*, i.e., perform a bulk insert of weather data into the database; 3. *download data* in a user-defined time window; 4. *register a station* which periodically pulls data into the database from the URL's weather station (the URL and the time interval are defined by the users at the registration time); 5. *query data from the database*, so the user can select stations, time window and weather dimension (e.g., temperature), and compute statistics on them.

The three-tier nature of the web-oriented systems requires a specific customization of the proposed oMVC pattern. Like MVC, oMVC entities must be implemented as elements constituting the final system architecture, whatever architectural style has been chosen (e.g., client-server, microservices, etc.). oMVC is in fact agnostic to the adopted architectural style, as it does not come with specific constraints to tackle different architectural styles. In the case of three-tier architectures, the state of an application should generally address several dimensions, each one associated with an application layer (presentation, application, and storage). In addition, implementing the storage tier by using a database introduces new challenges, as keeping data integrity between the database and a copy requires the adoption of specific solutions, and might yield inefficiencies.

We interpret observability in the selected application as follows. The State component consists of two distinct entities called, respectively, *WebAppState* and *StorageState*. The WebAppState component keeps track of the user requests that are generated in the presentation layer. Every request originates from an HTTPRequest issued by the user and ends when an HTTPResponse is sent back to the client. Every HTTPRequest/HTTPResponse is collected in the WebAppState in a map, and is associated with a unique identifier and a corresponding Action, that represents it in the application layer. Every HTTPRequest coming from the client is described by all its parameters, in order to allow for state reconstruction: they can include, for instance, the identifier if a remote station, the weather data that are transferred, etc. The StorageState component keeps track of all the SQL queries on the database, that are encoded as textual strings to guarantee the database reconstruction. Logging is carried out according to the rules described in Sec. 3, and is realized by recording tuples that include: a) the Action description, b) the HTTPRequest description,

[7] the full analysis tableau can be found online: [58]

```
@@BEGIN_LOG_ITEM@@
@@ACTION@@
CreateStationAction{data={"altitude":"1345",
                          "unitOfMeasure":{"rain":"mm/m2",...},
                          "latitude":"123","name":"Milan",
                          "type":"Country","longitude":"123"}}
@@PRE_TIMESTAMP@@ ...
@@PRE_STATE@@ HttpRequest: {...}
@@POST_STATE@@ Outcome: {"success": "true", "text": "..."}
QueryPerformed:
  insert into UnitOfMeasure (dewPoint, humidity, ...)
         values (NULL, 'g/m3', ...)
  insert into Station (altitude, latitude, ...)
         values (1345.0, 123.0, 123.0, 'Milan', 'Country', 1)
@@END_LOG_ITEM@@
```

FIGURE 4 A snippet of observability-enabled textual logs.

and c) the SQL query sent to the database. A snippet of a log is shown in Fig. 4.

Figure 5 shows a partial representation of the class diagram of the oMVC application, colored in black, and an overlay that identifies the fundamental roles of the two design patterns that underlie the oMVC architecture, namely, the MVC pattern and the Command pattern. The application exploits several View/Normal classes that are extensions of the Servlet class (which materialize the View of an MVC architecture). As an example, Fig. 5 shows the *UploadData* servlet, which handles the HTTP user request "Upload data". Every servlet requires the implementation of the method called *onWebAppStateChange()*, each one taking care of the construction of the HTTP response message specific to the associated HTTP user request. The *Controller* component is functionally realized by the Store component, which exposes a method called *propagateAction()* executing the logic that handles an HTTPRequest, captured by a servlet and represented by means of an Action object. Finally, the components WebAppState and StorageState impersonate the *Model* of the MVC pattern (Hibernate is also shown in the figure to clarify the presence of a component implementing persistence of data).

The state reconstruction can be realized by applying stepwise the operations that are specified in every log item on the WebAppState and StorageState components. In particular, the Action and the HTTPRequest can be used to reconstruct the maps in the WebAppState, while the queries can be executed incrementally to reconstruct the database, starting from an empty instance. (A detailed class and sequence diagram of the application are provided in the online Appendix [58] for the interested reader.)

5.3 | Metamodeling for Observability

To answer RQ0, hence to create a metamodel describing an observable architecture, we analyze oMVC in terms of its underlying design patterns and we identify the abstract roles that enable observability. In doing this exercise, we consider all the functional abstractions identified in Sec. 3.2 and applied to develop oMVC abstractions in Smarcello123 ec. 4.2. In addition, while analysing the abstractions required to materialise Event sourcing, we must also pay particular

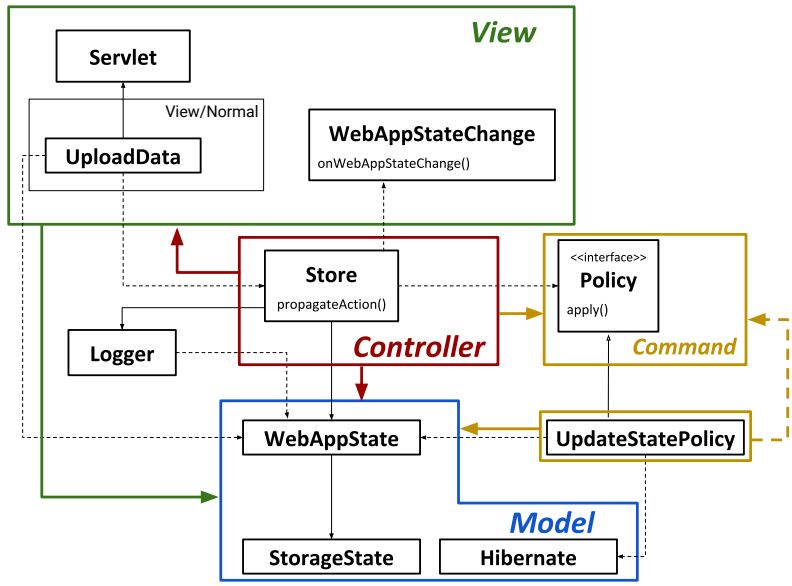
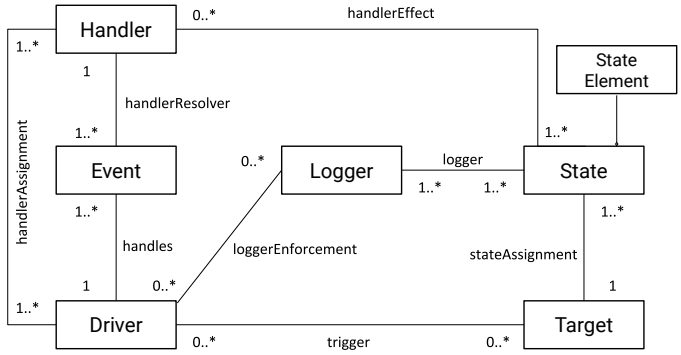


FIGURE 5 Underlying design patterns in the use case application.

attention to the role of the Command pattern that we use to separate the logic to elaborate an event from the enforcement of a state change. The result is the metamodel shown in Fig. 6.

We first identify an abstraction for each functional element in oMVC, and therefore assign a metaclass to State, Store, Policy, Action and Resolver. As explained in Sec. 4.2, Policies correspond to commands in the Command pattern and play a fundamental role in separating the logic for processing an event from the application of a state change is crucial in oMVC. Examining Fig. 5, we can see that the association which ties the MVC Controller with the Model and enables the effect of the controller's logic on the application's model, is refined in oMVC and transformed into an observable cause-effect relation. Policy objects make the cause-effect relation effectively concrete. This argument is general, and can be applied every time a cause-effect relation, that should be "observable", exists between two entities, one of which induces an effect on the other. Based on this analysis, we consider part of the oMVC metamodel the two abstractions that describe the roles in the Command pattern, namely Sender and Receiver. For the sake of uniformity with the oMVC nomenclature, we name the two abstractions Driver and Target. The Driver abstraction models Store in oMVC and Target is the abstraction for State. Their semantics reflects the meaning of Store and State in oMVC. A Driver component initiates a reaction to handle an application's event, and the consequence of managing the event determines an effect on a Target component, which undergoes modification accordingly. The oMVC Action is an instance of the general abstraction called Event. The occurrence of an Event implies that the logic implemented by a Handler is carried out to realize the effect on the Target. Handler is the abstraction of the oMVC Policy. The execution of the Handler's logic affects some State elements that contribute to the definition of the state of the Target. The modification realized by the Handler, as well as the event causing the change, are logged by storing the State element's value before and after the execution of the Handler's logic. Several relations exist among the aforementioned abstractions. A peculiar one is *handlerResolver* between Handler and Event which is made concrete through the oMVC Resolver. The other relations in the metamodel are the following.



(a) Metamodel for observable applications.

FIGURE 6 oMVC applied on weather forecast application and metamodel.

- *handlerAssignment* and *handles* become dependencies between, respectively, Store (Driver class) and the Policies (Handler classes) that manages events, and Store and Actions (Event classes).
- *handlerEffect* materializes through method *apply()* in Policy classes, whose purpose is to modify State (of a Target class) using a selected Policy (Handler class).
- *trigger* is reflected to method *propagateAction()* in Store (Driver class). This relation allows Store to apply a Policy (Handler class) to State (Target class) and possibly calculate a new application's state.
- *loggerEnforcement* and *logger* are dependencies between Store (Driver class) and Logger (Logger class), and between Logger and State (Target class).
- *State* includes public methods that can be used to manipulate the application's state or public attributes of the State (Target class). An attribute belongs to a target class through the *stateAssignment* relation. In essence, State represents the projection of the entire application's state that is observable.

6 | EVALUATION

6.1 | Costs of Observability by Design

Table 5 summarizes the FP counts, and connected variations of both the SOA and CS application, respectively (detailed analysis can be found in the online appendix [58]). As evident, there is a considerable difference between observable and unobservable architectures; more specifically, the observable version suffers from a double additional gain of costs in half of the systems function types (*Entry* and *Write* functions).

The analysis aims at evaluating the functional requirements FR1 and FR2 that we identified at the beginning of Sec. 4, and it has been carried out by considering the granularity at the class level, i.e., the classes are the atomic entities that interact with each other through methods. Since we limit the analysis to the functional requirements FR1 and FR2, the functional points evaluation can be restricted to the class Store and the class Logger. The functional users of the applications are the external "customers" of the application that play their role via Servlet components

	CS+MVC						SOA		
	Client side			Server side					
COSMIC FP	**Ob.**	**Un.**	**Var.**	**Ob.**	**Un.**	**Var.**	**Ob.**	**Un.**	**Var.**
Entry	58	32	+81	35	22	+50%	9	6	+50%
Exit	35	35	+0%	14	14	+0%	2	2	+0%
Read	1	0	-	1	0	-	2	1	+100%
Write	17	0	-	8	0	-	3	1	+150%

TABLE 5 COSMIC Function Point counts for observable (Ob.) and unobservable (Un.) version of the application, with evaluation of variance of observable with respect to unobservable architectures (Var.) across the types of data movement counts in the COSMIC taxonomy (col. 1).

in the SOA application, and via the client in the CS application; the edge that separates the external environment from the application materializes through the Servlets and the clients. For this reason, only Store and Logger directly perform a function to the user of the application while Action, Resolver and Policy contribute indirectly. Resolver and Policy are functional with respect to Store while Action is a representation of an event and therefore can be equated more to datum than a functional element.

The Persistent storage consists of the application log in both the architectural styles and includes also the database in the SOA application. Since the two versions (observable/non-observable) of the application provide a different set of functionalities to the user (i.e. the functionalities that concretely realise the FR1 and FR2 requirements and thus the observability) it is correct to expect (and see) that the number of function points in both CS or SOA architectural solutions, show a higher value in the observable versions.

The functional processes that are directly mapped to the functional requirements FR1 and FR2 are: log recording, database/state reading/writing and database/state reconstruction. The mapping between the functional requirements and the processes is defined as follows: FR1 implies log recording, database(SOA)/state(CS) reading/writing, and database(SOA)/state(CS) reconstruction; while FR2 implies database(SOA)/state(CS) reading/writing.

It should be noted that our counts and the simplistic nature of our case-study make these estimations very optimistic, and therefore the connected figures are likely to be much higher in a more complex architecture. In summary, in the scope of RQ1, the following results:

Observability Costs. Observable architectures are bound to be more costly to design and implement; conservatively, increasing costs can amount to at least a factor >0.5 in at least 50% of the COSMIC function types.

Consequently, this finding indicates that a new architecture trade-off analysis [61] needs to be instrumented which features of service-level agreements [62] such as service-disruption costs, service discontinuity times [62] with the dimensions connected to observability, costs, and required efforts.

6.2 | Impacts of Observability-By-Design

To answer RQ2 we first present in Table 6 an overview of the Lines of Code (LOC) of both case studies, CS-based and SOA-based, showing the additional effort required to implement the observability-by-design pattern. While in the CS case study the additional effort is less than 40% and in line with the expectation observed in Table 5, for the SOA

	CS-Server side	CS-Client side	SOA
MVC	1920	2122	754
oMVC	2437	2897	1461
% diff	26.9	36.5	93.7

TABLE 6 Lines of Code.

application the effort increase is close to 100%. This difference can be explained by the distributed nature of the SOA application requiring extra effort to add observability features.

Maintainability

Using CodeMR we have then analysed both case studies in terms of well-known software architecture metrics, i.e., *Complexity*, *Size*, *Coupling* and *Lack of cohesion*. We found our analysis on those metrics as they can be used as a significant measure of the effort needed to maintain a software artifact.

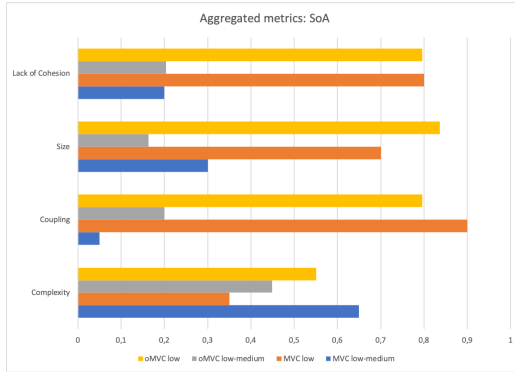


FIGURE 7 Aggregated Metrics: The SOA case.

In this respect, we recall the definition of maintainability that has been adopted by ISO 9126 [63, 64]: "the ease with which a software system or component can be modified to correct faults, improve performance or other attributes, or adapt to a changed environment [. . .]" Figs 7 and 8 show the relative percentages of classes with metrics' values low and low-medium(-high) implemented in both the CS and SOA applications, for the observable and the standard patterns. We can compare results for *Lack of cohesion* metric in the SOA case, in both the measures; whereas, we observe a slightly higher percentage of classes with low values in the CS case. For *Size* metric, we observe a slightly higher percentage of classes with low-medium value for the standard pattern in the SOA case, while all the other percentages are comparable. A specular situation can be observed for *Coupling*, while for *Complexity*, we observe in both applications that the standard pattern presents a higher percentage of classes with medium values.

Observability vs. Maintainability. Changing the case study and the architectural pattern *does* indeed impact the effort needed to assure observability in terms of LOC, but **does not** impact significantly on maintainability.

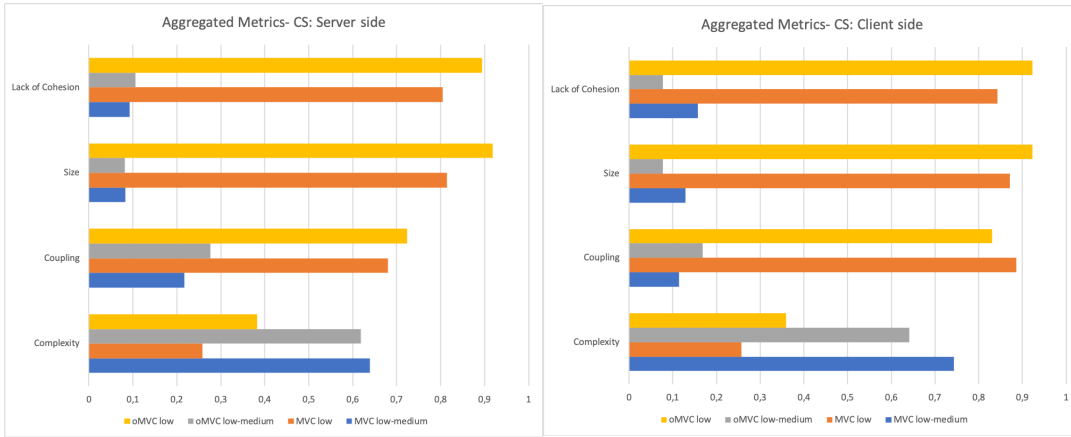


FIGURE 8 Aggregated Metrics: the CS case.

A careful analysis of our proposed architecture points out that the following two qualities of Maintainability are mainly eased by an observable architecture. The first quality is the *Analizability* which consists of "the aspects that help predict the maintainer or user's spent effort (or resources) in trying to diagnose deficiencies and causes of failure or for identifying parts to be modified in the system". In this scope, an observable architecture fosters *Activity recording* ("a measure of how thoroughly the system status is recorded"), *Audit trail capability*, *Failure analysis capability* and *Status monitoring capability* ("a measure of how easy it is, of the ability for a user (or maintainer) to identify the specific operation that caused a failure, and to get monitored data for operations that cause failures during the actual operation of the system"). The isolation of the entity State, which captures the (projection of) application's state, and the Logger abstraction, which provides the functionality for recording a declarative representation of it every time an action is triggered, are essential to allow for improving these aspects. In addition, the availability of a representation of a cause-effect relation between changes of the State and Actions, reduces the effort needed to analyze the application runtime, hence simplifying error identifications and problem diagnosis. Given the nature of actions, that is, "messages" signaling the occurrence of an internal or external relevant event, and given that every Action is handled by using suitable policies, application failures can be easily mapped to Actions, and error handling procedure be implemented through specific policies. Getting data that concerns failures is as doable as defining suitable actions that identify architecture failures, and that are enriched with runtime information about them (actions can convey contextual data).

Changeability

The second quality concerns *Changeability* which is "the aspects that help predict the maintainer's or user's effort when trying to implement a specified modification to the system". In particular, the *Modification complexity* is the reference quality in the evaluation as it "measures how easily a maintainer can change the software". Observable architectures impose a precise scheme for implementing the business logic of an application and the use of a precise set of abstractions. These abstractions have precise and unambiguous definitions and follow a fixed structure, so that the developer interacts with them predictably. A rigorous relationship between the business logic and the abstractions required to implement it eases the maintenance efforts needed for the architecture and, hence, improves the overall maintainability of the software product.

Finally, the last quality concern to mention is *Reusability* (considered in ISO/IEC 25010) which is "the ability of the software components to be used in developing more than one software product or other assets". Adopting patterns in the architecture definition promotes component reuse, especially for components implementing specific functionalities of Policies through the Command pattern. Consider, for instance, Policies for handling CRUD operations on databases, such as UpdateStatePolicy in our oMVC implementation, or data statistics, temporal series analysis, predictive functionalities, etc.

6.3 | Observability vs. Traceability

Traceability is not defined by ISO 9126, but a definition can still be found in the software engineering community. Two are the meaning that are commonly used to shape the notion of traceability. In software development, traceability is "the extent to which a team of developers can trace work items across the software development lifecycle (SDLC)". In essence, traceability enables teams to keep track of what is currently under development and what has already been done. Conversely, a second sense is traceability in software testing. In this context, traceability focuses on tracing tests, test cases, and results. Based on the common two interpretations of traceability in software engineering, we can claim that our definition of observability is not an alternative to the common interpretation, rather it is transversal to it, being ultimately a novel concept.

6.4 | Threats to validity

Concerning standard threats to validity frameworks such as Wohlin et al. [65], we in fact encountered and tried to address several internal and construct validity threats, especially in the scope of the use of COSMIC FP Counting to address RQ1 or within our case-study. On the one hand, we systematically used Inter-Rater reliability assessment methods every time conclusion validity was at stake, and what is more, we triangulated also the designers and implementors of the case-study at hand, to avoid observer bias and similar circumstances. At the same time, the size and complexity of the prototype case under study was deliberately controlled to ensure an extent of external validity. We consider the simplistic case under study in this paper as an optimistic lower-bound for whatever external considerations should practitioners do regarding to their own practice. Beyond these, a number of additional issues and scope limitations are reported below.

Lack of quantitative metrics for observability

Although we offered a concise definition of architecture observability, the major limitation connected to it is that observability cannot yet be evaluated numerically as there is not a quantitative architecture-based metric that allows designers to estimate and detect the observability of an implementation upfront—other than designing for it explicitly as we did—and hence to trade-off over different design options. Again, the effects of observability can only be measured indirectly, for instance, one can refer to the standard ISO 9126 [64] (like the ones implemented in CodeMR) for the evaluation of some software qualities that are affected by the implementation of a specific architecture. This limitation we encountered in previous work still applies, although we plan to address it more head-on with results stemming from the formalization and experimentations reported in this paper.

Implementation complexity

The conclusions and the graphs provided in the previous sections are in line with our previous work and again remark a generalized negative impact of observability in the lines of code, while for most of the considered metrics the overall

average behavior is comparable. This is again not surprising, however, it remarks that architecture styles do not play a role in mediating system observability. More specific measurements for architecture-level observability are in order to proceed in defining and exploring such architecture property.

Proof-of-concept evaluation

As previously stated, the evaluation of the observability in this work has been carried out through an experimental study involving a proof-of-concept case-study of limited size and controlled characteristics; this focus selection reflects the nature of our target software constructs, which are relatively simple and small, therefore there is a limited external validity for this study while measurements should apply to several studies to be conclusive. This avenue deserves further exploration and is planned as future work. More specifically, although in this paper we aimed at offering a slightly deeper analysis beyond previous work and to compare different implementations of more significant but still small and manageable applications. In the future, we plan to enlarge the size and complexity of targeted applications and a larger dataset, focusing on medium and large applications hosted on GitHub.

7 | CONCLUSIONS

Modern software systems feature unprecedented scales and complexity. A way to design for such complexity while coping with the required service continuity aspects around such complex applications' use is the concept of observability. Designing for it can leverage on event sourcing, i.e., capturing all changes to an application state as a sequence of events, which could in principle be traced and subsequently analysed. In layman's terms, observability reflects the extent to which a complex software application can manifest its internal workings and operations to the outside world (e.g., to a specific monitoring technology stack), with the possibility of verifying—through runtime- verification and trace-checking—specific service continuity or quality properties. To progress in the definition and operationalisation of observability at an architecture level, we: (1) provide for a rigorous definition for it and elaborate its required design principles; (2) understand its impact on systems architecture maintenance costs. Our results show that observability yields an additional effort regarding conventional implementations and mildly affect maintenance and evolvability of software architectures. The conclusion here is that Event sourced observability is in fact a worthwhile investment.

In the future, we aim at specific metrics designed to support observability not only as a binary value but also as a fuzzified metric to further explore the premises and consequences of observability-by-design. At the same time, we aim to investigate runtime model-checking facilities compatible with the notions and scope defined in this paper.

references

[1] Google Chrome Team, The Evolution of the Web;. http://www.evolutionoftheweb.com/.

[2] Aagedal JØ. Quality of Service Support in Development of Distributed Systems. PhD thesis, University of Oslo; 2001.

[3] Reale A. Quality of Service in Distributed Stream Processing for large scale Smart Pervasive Environments. PhD thesis, University of Bologna, Italy; 2014.

[4] Horvat G, Zagar D, Vlaovic J. Evaluation of quality of service provisioning in large-scale pervasive and smart collaborative wireless sensor and actor networks. Advanced Engineering Informatics 2017;33:258–273.

[5] Logstash; 2022. https://www.elastic.co/logstash/.

[6] Dynatrace; 2022. https://www.dynatrace.com/.

[7] Splunk; 2022. https://www.splunk.com/.

[8] Grafana; 2022. https://grafana.com/.

[9] Apache Skywalking; 2022. https://skywalking.apache.org/.

[10] Muller E, Monitoring and Observability; 2018. https://theagileadmin.com/2018/02/16/monitoring-and-observability/.

[11] Splunk, What is Observability; 2021. https://www.splunk.com/en_us/data-insider/what-is-observability.html.

[12] Bass L, Clements P, Kazman R. Software Architecture in Practice. 3rd ed. Addison-Wesley Professional; 2012.

[13] Erb B, Kargl F. Combining Discrete Event Simulations and Event Sourcing. In: Proceedings of the 7th International Conference on Simulation Tools and Techniques - SIMUTools. Brussels, Belgium: Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering; 2014. p. 51–55.

[14] Han S, Choi JI. V2X-Based Event Acquisition and Reproduction Architecture with Event-Sourcing. In: Proceedings of the 6th International Conference on Computing and Data Engineering - ICCDE. New York, NY, USA: Association for Computing Machinery; 2020. p. 164–167.

[15] Overeem M, Spoor M, Jansen S. The Dark Side of Event Sourcing: Managing Data Conversion. In: Proceedings of the 24th International Conference on Software Analysis, Evolution and Reengineering - SANER. Washington, DC, USA: IEEE Computer Society Press; 2017. p. 193–204.

[16] Bass L, Weber I, Zhu L. DevOps: A Software Architect's Perspective. SEI Series in Software Engineering, New York, NY, USA: Addison-Wesley; 2015.

[17] Bersani MM, Bianculli D, Ghezzi C, Krstić S, San Pietro P. Efficient Large-Scale Trace Checking Using Mapreduce. In: Proceedings of the 38th International Conference on Software Engineering - ICSE New York, NY, USA: Association for Computing Machinery; 2016. p. 888–898.

[18] Ganter B, Wille R. Formal Concept Analysis: Mathematical Foundations. Berlin, Heidelberg: Springer-Verlag; 1999.

[19] Schewe KD, Thalheim B. Conceptual Modelling of Web Information Systems. Data & Knowledge Engineering 2005;54(2):147–188.

[20] Akkermans H, Speel PH, Ratcliffe A. Problem, Opportunity, and Feasibility Analysis for Knowledge Management: An Industrial Case Study. In: Proceedings of the 12th Workshop on Knowledge Acquisition, Modeling and Management - EKAW. Berlin, Heidelberg: Springer-Verlag; 1999. p. 147–188.

[21] Burbeck S, Applications Programming in Smalltalk-80(TM): How to use Model-View-Controller (MVC); 1987. http://st-www.cs.uiuc.edu/users/smarch/st-docs/mvc.html.

[22] Richards M, Microservices vs. Service-Oriented Architecture. O'Reilly Media, Inc.; 2016. https://www.oreilly.com/learning/microservices-vs-service-oriented-architecture.

[23] Tamburri DA, Bersani MM, Mirandola R, Pea G. DevOps Service Observability By-Design: Experimenting with Model-View-Controller. In: Proceedings of the 7th European Conference on Service-Oriented and Cloud Computing - ESOCC, vol. 11116 of Lecture Notes in Computer Science Berlin, Heidelberg: Springer-Verlag; 2018. p. 49–64.

[24] Abualkishik AZ, Ferrucci F, Gravino C, Lavazza L, Liu G, Meli R, et al. A study on the statistical convertibility of IFPUG Function Point, COSMIC Function Point and Simple Function Point. Information and Software Technology 2017;86(C):1–19.

[25] Ibrahim A. Prediction of Quality of Service of Software Applications. In: Proceedings of the 5th European Conference on Service-Oriented and Cloud Computing - ESOCC, vol. 707 of Communications in Computer and Information Science Berlin, Heidelberg: Springer-Verlag; 2016. p. 260–273.

[26] Toffetti G, Brunner S, Blöchlinger M, Dudouet F, Edmonds A. An architecture for self-managing microservices. In: Proceedings of the 1st International Workshop on Automated Incident Management in Cloud - AIMC. New York, NY, USA: Association for Computing Machinery; 2015. p. 19–24.

[27] Galletta A, Carnevale L, Buzachis A, Celesti A, Villari M. A Microservices-Based Platform for Efficiently Managing Oceanographic Data. In: Proceedings of the 4th International Conference on Big Data Innovations and Applications - Innovate-Data. Washington, DC, USA: IEEE Computer Society Press; 2018. p. 25–29.

[28] Shahin M, Zahedi M, Babar MA, Zhu L. An Empirical Study of Architecting for Continuous Delivery and Deployment. Empirical Software Engineering 2019;24(3):1061–1108.

[29] Hu K, Data Observability vs. Software Observability; 2021. https://www.metaplane.dev/blog/data-observability-vs-software-observability.

[30] Google, DevOps measurement: Monitoring and observability; 2022. https://cloud.google.com/architecture/devops/devops-measurement-monitoring-and-observability.

[31] Carey S, What is observability? Software monitoring on steroids; 2021. https://www.infoworld.com/article/3607980/what-is-observability-software-monitoring-on-steroids.html.

[32] Kalman RE. Mathematical Description of Linear Dynamical Systems. Journal of the Society for Industrial and Applied Mathematics Series A Control 1963;1(2):152–192.

[33] Eventuate, Solving distributed data management problems in a microservice architecture; 2021. https://eventuate.io/.

[34] Fielding RT. Architectural Styles and the Design of Network-based Software Architectures. Doctoral dissertation, University of California, Irvine; 2000.

[35] Andrews GR. Paradigms for Process Interaction in Distributed Programs. ACM Computing Surveys 1991;23(1):49–90.

[36] Taylor RN, Medvidovic N, Anderson KM, Whitehead EJ, Robbins JE, Nies KA, et al. A component- and message-based architectural style for GUI software. IEEE Transactions on Software Engineering 1996;22(6):390–406.

[37] Cooper K, Peters T, The MVVM pattern; 2012. https://msdn.microsoft.com/en-us/library/hh848246.aspx.

[38] Taligent I, MVP: Model-View-Presenter; 1996. http://www.wildcrest.com/Potel/Portfolio/mvp.pdf.

[39] Fowler M, Event Sourcing; 2021. https://martinfowler.com/eaaDev/EventSourcing.html.

[40] Narumoto M, Buck A, Peterson N, Kshirsagar D, Coulter D, Kittel C, et al., Event Sourcing Pattern; 2021. https://docs.microsoft.com/en-us/azure/architecture/patterns/event-sourcing.

[41] Richardson C, Pattern: Event Sourcing; 2021. https://microservices.io/patterns/data/event-sourcing.html.

[42] Narumoto M, et al, Command Query Responsibility Segragation; 2021. https://docs.microsoft.com/en-us/azure/architecture/patterns/cqrs.

[43] Fowler M, Command Query Responsibility Segragation; 2021. https://martinfowler.com/bliki/CQRS.html.

[44] Debski A, Szczepanik B, Malawski M, Spahr S, Muthig D. A Scalable, Reactive Architecture for Cloud Applications. IEEE Software 2018;35(2):62–71.

[45] Atlee JM, Gannon JD. State-Based Model Checking of Event-Driven System Requirements. IEEE Transaction of Software Engineering 1993;19(1):24–40.

[46] Kiczales G, Lamping J, Mendhekar A, Maeda C, Lopes CV, Loingtier J, et al. Aspect-Oriented Programming. In: Proceeding of the 11th European Conference on Object-Oriented Programming - ECOOP, vol. 1241 of Lecture Notes in Computer Science Berlin, Heidelberg: Springer-Verlag; 1997. p. 220–242.

[47] Microsoft, Azure; 2021. https://azure.microsoft.com.

[48] Apache, Apache Kafka; 2021. https://kafka.apache.org/.

[49] Eventuate, Eventuate Local; 2021. https://github.com/eventuate-local/eventuate-local/.

[50] Taylor RN, Medvidovic N, Dashofy EM. Software Architecture: Foundations, Theory, and Practice. Wiley Publishing; 2009.

[51] Paolone G, Marinelli M, Paesani R, Felice PD. Automatic Code Generation of MVC Web Applications. Computers 2020;9(3):56.

[52] Huang Z, Liang Y. Research of data mining and web technology in university discipline construction decision support system based on MVC model. Library Hi Tech 2020;38(3):610–624.

[53] Avizienis A, Laprie J, Randell B, et al. Fundamental concepts of dependability. University of Newcastle upon Tyne, Computing Science; 2001.

[54] Gamma E, Helm R, Johnson R, Vlissides J, Design Patterns: Elements of Reusable Object Oriented Software; 1995.

[55] Capers J. Function points as a universal software metric. ACM SIGSOFT Software Engineering Notes 2013;38(4):1–27.

[56] Hira A, Boehm BW. Using Software Non-Functional Assessment Process to Complement Function Points for Software Maintenance. In: Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement - ESEM. New York, NY, USA: Association for Computing Machinery; 2016. p. 1–6.

[57] Krippendorff K. Content Analysis: An Introduction to Its Methodology (fourth edition). Sage Publications; 2018.

[58] Appendix of the paper, April 2022; April 2022. https://www.dropbox.com/sh/2hdvl3o3qzzik6c/AAC4so_ZcNXZrEdoJXZ3Nv6Sa?dl=0.

[59] Gui G, Scott PD. Measuring Software Component Reusability by Coupling and Cohesion Metrics. Journal of Computers 2009;4(9):797–805.

[60] Dhama HS. Quantitative Models of Cohesion and Coupling in Software. Journal of Systems and Software 1995;29(1):65–74.

[61] Bellomo S, Gorton I, Kazman R. Toward Agile Architecture: Insights from 15 Years of ATAM Data. IEEE Software 2015;32(5):38–45.

[62] Ghezzi C, Guinea S. Run-Time Monitoring in Service-Oriented Architectures. Test and Analysis of Web Services 2007;p. 237–264.

[63] IEEE Standard Glossary of Software Engineering Terminology. IEEE Std 61012-1990 1990 12;p. 1–84.

[64] Software engineering – Product quality – Part 1: Quality model. Geneva: International Organization for Standardization; 2001.

[65] Wohlin C, Runeson P, Höst M, Ohlsson MC, Regnell B. Experimentation in Software Engineering. Berlin, Heidelberg: Springer-Verlag; 2012.

[66] Abuosba KA, El-Sheikh AA. Formalizing Service-Oriented Architectures. IT Professional 2008;10(4):34–38.

[67] W3C Working Group Note 11: Web Services Architecture, May 2020; 2020. http://www.w3.org/TR/ws-arch//#stakeholder.

Appendix

The study has been carried out by considering an application that supplies services through a web platform. A Service Oriented Architecture (SOA) is software design style that requires applications to provide services through a communication protocol over a network. A Web service is a special implementation case of SOA that achieves most of the architecture's properties by deploying decoupling mechanisms [66]. A Service Oriented Architecture, as defined by the W3C Consortium [67], is a distributed system characterized by the following properties:

- *Logical view*: The service is an abstraction of actual programs, databases, business processes, etc., defined in terms of its role.
- *Message orientation*: The service is defined in terms of the messages exchanged between provider agents and requester agents, and not the properties of the agents themselves. The agent is the concrete piece of software or hardware that sends and receives messages.
- *Description orientation*: A service is described by machine-processable meta data. The description supports the public nature of the SOA: only those details that are exposed to the public, and important for the use of the service, should be included in the description.
- *Granularity*: Services tend to use a small number of operations with relatively large and complex messages.
- *Network orientation*: Services tend to be oriented toward use over a network, though this is not an absolute requirement.
- *Platform neutral*: Messages are sent in a platform-neutral, standardized format delivered through the interfaces. XML is the most obvious format that meets this constraint.

The application has been designed in order to satisfy the previous properties, by adopting a three-tier architecture and a client/server communication schema. In particular, the application has been implemented using J2EE, coupled with Apache Tomcat[8] as a web server (an open source software implementation of the Java Servlet and Java Server Pages technologies). In order to map entities in the persistence layer to objects in the application, we used the standard Java Bean naming conventions. Moreover, we employed Hibernate[9] to solve the object/relational impedance mismatch. The persistence layer of the application is realized through a MySQL database. Considering this setting, the application communicates with the clients via HTTP requests and with database via SQL queries.

8 | UML CLASS DIAGRAM OF THE OMVC VERSION

The class diagram modeling the implemented application is shown in Fig. 9. The relations among the classes are those explained in Sec. 4 and shown in Fig. 3, while the implementation of the View/Normal components is infrastructure-specific, as it leverages the Java Servlet technology offered by the J2EE framework. Servlets offer built-in functionalities for managing the full interaction between the application layer and the presentation layer, and therefore they turned out to be a suitable means to implement View/Normal components of the application. Every View/Normal component is a Servlet that implements the interface onWebAppStateChange, and realizes an Observer pattern along with the

[8] http://tomcat.apache.org/

[9] https://hibernate.org/orm/

WebAppState component. By means of WebAppStateChange, every View/Normal component can be kept updated as soon as a change in the State component occurs.

The class WebAppState includes the following attributes:

requests. It maps all the HTTPRequests issued by the clients to the corresponding numerical identifier.
responses. It maps all the HTTPResponses sent to the clients to the corresponding numerical identifier.
actions. It is a map that stores all the Actions issued by the clients. Every request is identified by a long value.
WebAppState. the attribute is used to force the uniqueness of the instance, as object WebAppState is a singleton.

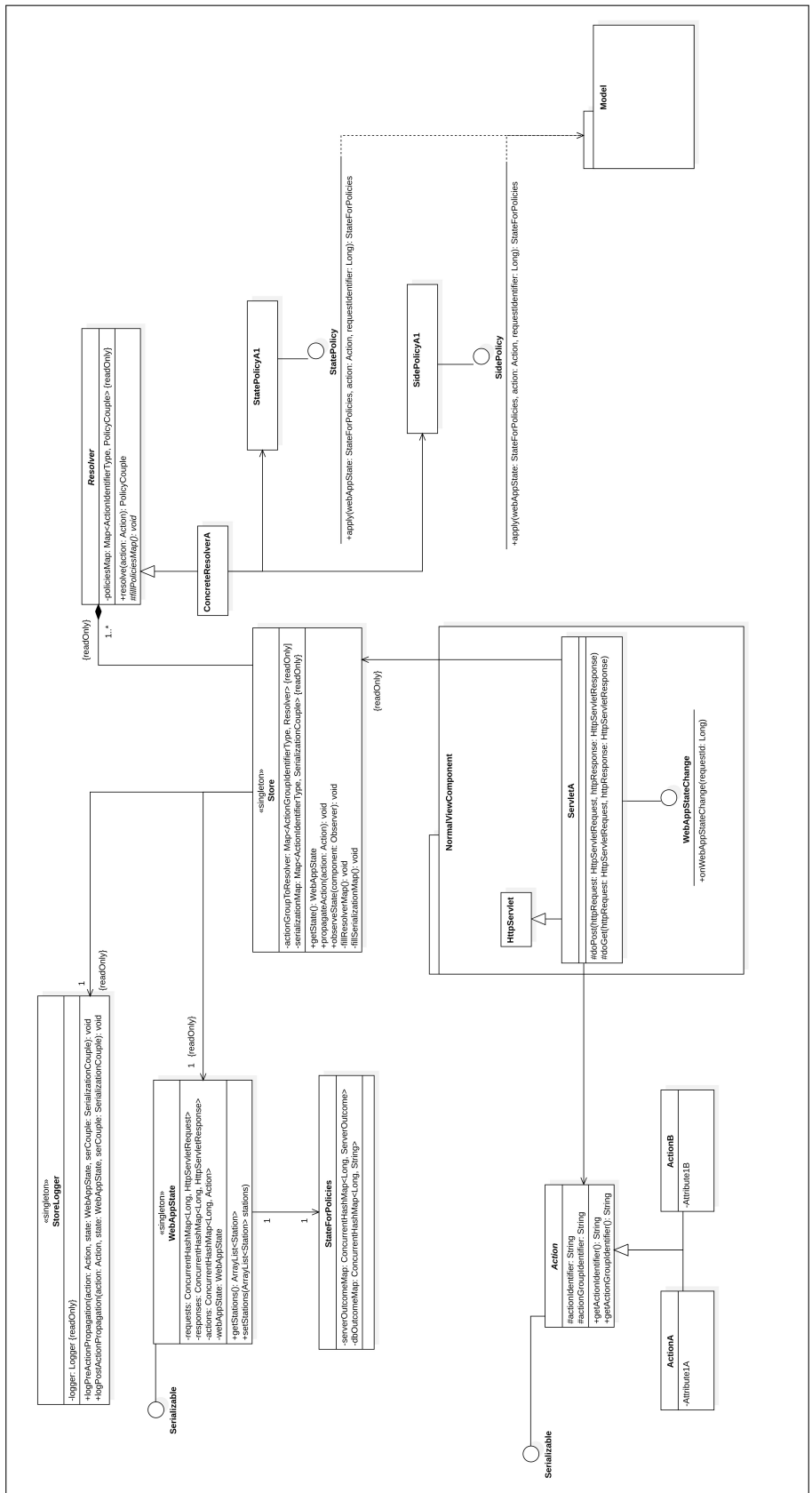
The class StateForPolicies is called in the paper "StorageState", and includes the following attributes:

serverOutcomeMap. It maps a numerical identifier associated with an Action to the server outcome, that is described by an instance of class ServerOutcome. ServerOutcome is a wrapper for a generic response, and it is mainly used, in this implementation, to hide Strings.
dbOutcomeMap. Similarly to the previous attribute, dbOutcomeMap is a collection of outcomes produced by the external database.

The application behavior is founded on five fundamental Actions, that reflect the functionalities exposed to the client.

CreateStationAction. It is associated with an HTTPRequest (POST) issued by the client that creates a new weather station in the database. An object includes the following information: altitude, unitOfMeasure (rain, windModule, temperature, humidity, pressure, windDirection, dewPoint), latitude, name, type and longitude.
DownloadDataAction. It is associated with an HTTPRequest (GET) issued by the client that retrieves data measured by a given weather station and within a specific time frame defined by an initial and final timestamps. The action generates an HTTPResponse with a payload that list all the measurements taken by the station in the specified time frame.
QueryDataGraphAction. It is similar to CreateStationAction but the selection is limited to a specific dimension (e.g., humidity).
RegisterStationAction. It is associated with an HTTPRequest (POST) issued by the client that enables the server to pull data from a given URL associated with a weather station.
UploadDataAction. It is associated with an HTTPRequest (POST) issued by the client and enables a manual insertion of weather data from the uses.

For the sake of conciseness, the class diagram does not show the classes that model the information exchange by the application with the database and the external environment, and that are related to the weather station and data. They are the same as those shown in Fig. 11 in the package *model*.


«singleton»
StoreLogger
-logger: Logger {readOnly}
+logPreActionPropagation(action: Action, state: WebAppState, serCouple: SerializationCouple): void
+logPostActionPropagation(action: Action, state: WebAppState, serCouple: SerializationCouple): void
Serializable
«singleton»
WebAppState
-requests: ConcurrentHashMap<Long, HttpServletRequest>
-responses: ConcurrentHashMap<Long, HttpServletResponse>
-actions: ConcurrentHashMap<Long, Action>
-webAppState: WebAppState
+getStations(): ArrayList<Station>
+setStations(ArrayList<Station> stations)
StateForPolicies
-serverOutcomeMap: ConcurrentHashMap<Long, ServerOutcome>
-idOutcomeMap: ConcurrentHashMap<Long, String>
Resolver
-policiesMap: Map<ActionIdentifierType, PolicyCouple> {readOnly}
+resolve(action: Action): PolicyCouple
#fillPoliciesMap(): void
ConcreteResolverA
StatePolicyA1
StatePolicy
+apply(webAppState: StateForPolicies, action: Action, requestIdentifier: Long): StateForPolicies
SidePolicyA1
SidePolicy
+apply(webAppState: StateForPolicies, action: Action, requestIdentifier: Long): StateForPolicies
Model
«singleton»
Store
-actionGroupToResolver: Map<ActionGroupIdentifierType, Resolver> {readOnly}
-serializationMap: Map<ActionIdentifierType, SerializationCouple> {readOnly}
+getState(): WebAppState
+propagateAction(action: Action): void
+observeState(component: Observer): void
-fillResolverMap(): void
-fillSerializationMap(): void
NormalViewComponent
HttpServlet
ServletA
#doPost(HttpRequest: HttpServletRequest, httpResponse: HttpServletResponse)
#doGet(HttpRequest: HttpServletRequest, httpResponse: HttpServletResponse)
WebAppStateChange
+onWebAppStateChange(requestId: Long)
Action
#actionIdentifier: String
#actionGroupIdentifier: String
+getActionIdentifier(): String
+getActionGroupIdentifier(): String
ActionB
-Attribute1B
ActionA
-Attribute1A
Serializable
{readOnly}
1..*
1
1 {readOnly}
1
1
{readOnly}
{readOnly}

9 | UML SEQUENCE DIAGRAM OF THE OMVC VERSION

The sequence of operations that are performed upon user requests is depicted in the sequence diagram in Fig. 10. The steps are the followings:

- The HTTPServlet first captures an HTTPRequest (1), and then it instantiates the proper Action (2) based on the requested functionality (that includes all the parameter values required to carry out the operation). Every HTTPServlet realizes a View/Normal component, and for this reason, the servlets call the method propagateAction of the Store component (3).
- the Store component takes care of the execution of the Action by calling the Resolver component (5) to retrieve the suitable Policy that can manage the request. To this end, the Store component first select the proper class of Resolvers that can include the one able to deal with the current Action (4).
- The selected Resolver instantiates the Policy that can manage the Action (6), and returns it to the Store component (7).
- The Store component can now process the user request as follows:
  - it requests the Logger component to write the projection of the application state to the log (8,9) before the Action takes place. In our implementation, this amounts to record the served HTTPRequest that is stored in attribute “requests” of WebAppState and the associated Action (with all the parameters).
  - it calls the method “apply” that (all) Policies implement to realize the transition of the application state (10,11). The Policy actually carries out the computation enforcing the change of state (12,13).
  - it requests the Logger component to write the projection of the application state to the log (14,15) after the execution of the Action. In our implementation, this amounts to record the outcome of the overall Action and the outcome of the database manager that performed the operation defined by the executed Policy (14,15).
  - finally, it notifies View/Normal components that the application state has been changed. Notifications are materialized through the method onWebAppStateChange, that is implemented in the Servlet implementing the View/Normal components.

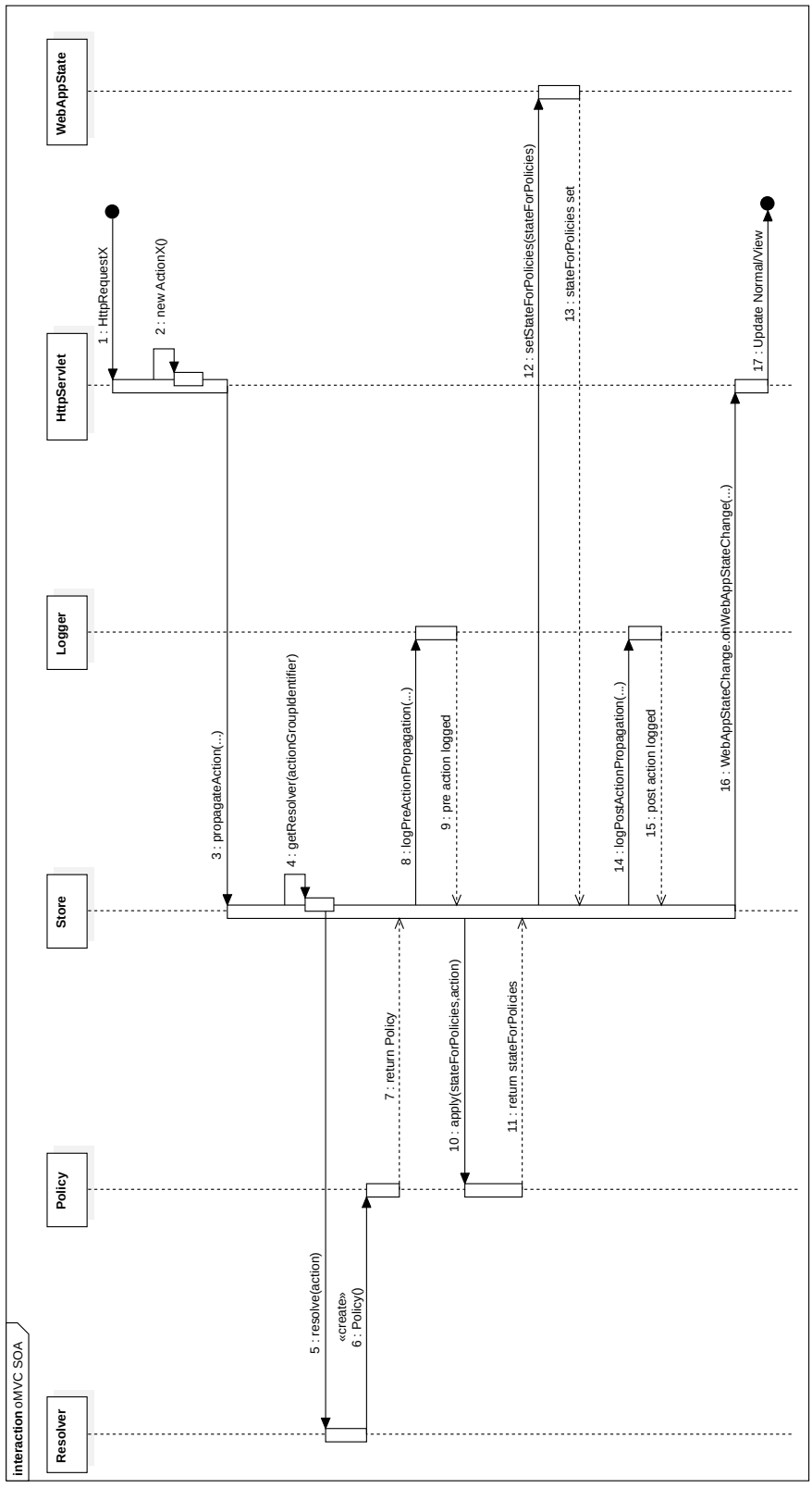


FIGURE 10 Sequence diagram of the oMVC version

10 | UML CLASS DIAGRAM OF THE MVC VERSION

Figure 11 shows the UML class diagram of the MVC version of the application. The package *model* includes all the classes that represent the information stored in the database. The same are used in the oMVC version.

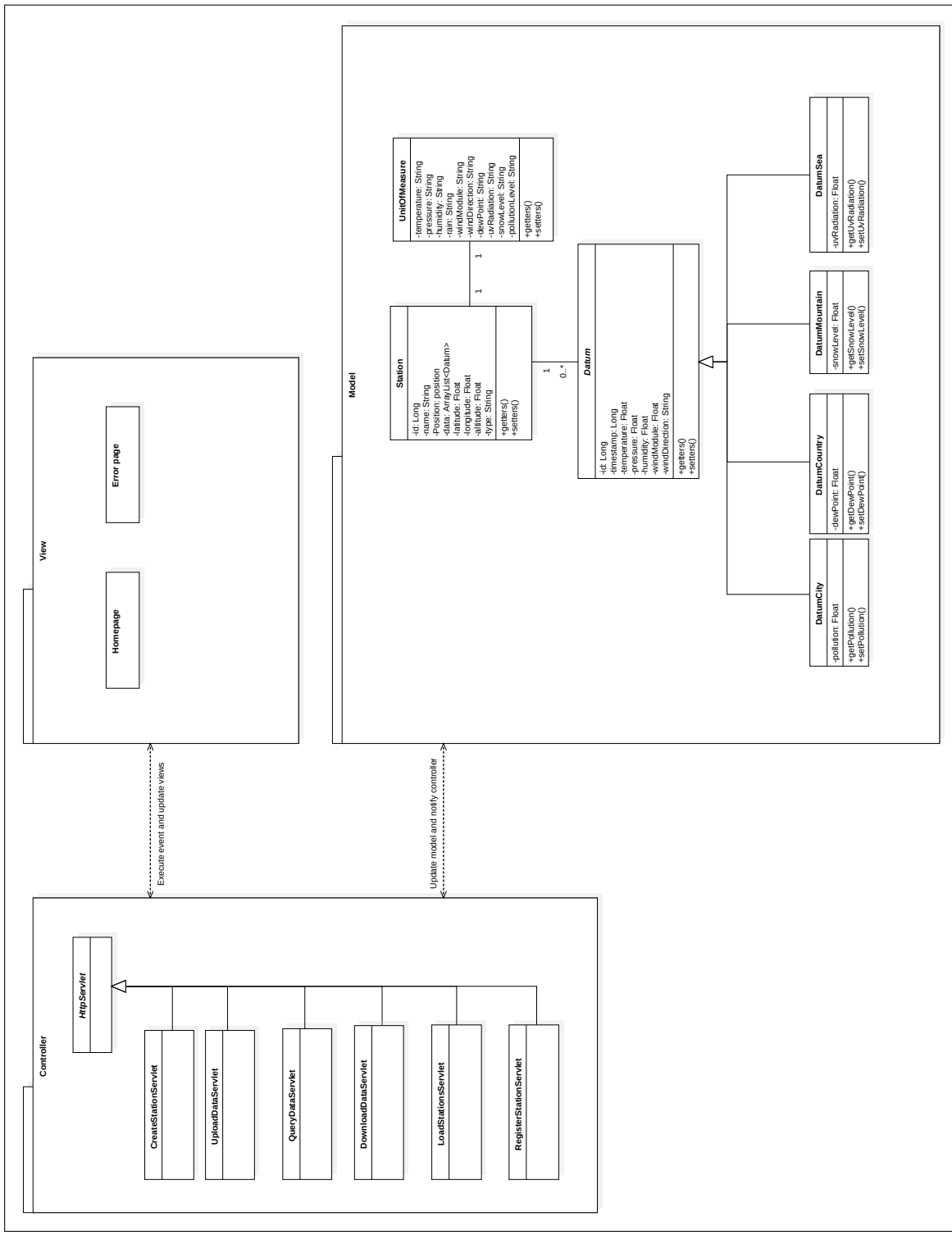


FIGURE 11 Class diagram of the MVC version

11 | FUNCTION-POINT ANALYSIS

We summarize the analysis of functional points carried out for the MVC/oMVC version of the application. According to the CORDIS workflow, we unwind the following steps.

11.1 | oMVC version.

Purpose.
The analysis aims at evaluating the functional requirements FR1 and FR2 that we identified in Sec. 4.

Granularity.
We conduct the analysis by considering the granularity at the class level, i.e., the classes are the atomic entities that interact with each other through methods that determine their behavior; and based on their methods, the analysis is carried out.

Scope.
Since we limit the analysis to the functional requirements FR1 and FR2, the scope of the functional points evaluation can be restricted to the class Store and the class Logger.

Functional users.
The functional users of the application are external "customers" of the application that play their role by means of Servlet components, and GET/POST methods. Equivalently, we can even consider Servelts as functional users, as they are the "proxy" of the users.

Boundary.
The edge that separates the external environment from the application, i.e., the application and the storage layer, materialize through the Servlets components, in the presentation layer.

Persitent storage.
It consists of two distinct entities, that are the database and the application log.

Functional processes.
The functional processes that are directly mapped to the functional requirements FFR2 are the following: log recording, database reading/writing and database reconstruction. The mapping between the functional requirements and the processes is defined as: 1. FR1 implies log recording, database reading/writing, and database reconstruction; 2. FR2 implies database reading/writing.

Triggering events.
The following events activate functional processes by means of suitable Actions in the application:

data insertion (through POST events): CreateStationAction, UploadDataAction, and RegisterStationAction

data collection (through GET events): DownloadDataAction, LoadMinimizedStationsAction, QueryDataGraphAction.

Data groups.
For every functional process, the corresponding COSMIC actions are identified.
log recording

- Entry: through a POST event and by means of CreateStationAction, UploadDataAction, and RegisterStationAction (associated to State Policies - so, they are logged).
- Exit: no exit, as log recording does not produce outcomes for a functional user.
- Read: no read, as this process is a writing.
- Write: this process is a writing.

database reading

- Entry: through a GET event and by means of DownloadDataAction, LoadMinimizedStationsAction, and QueryDataGraphAc (associated to Side Policies - so, they are not logged).
- Exit: it is a cvs file including selected data. It occurs through an HTTPResponse.
- Read: it is achieved through SELECT queries in the database.
- Write: no write, as this process is a reading.

database writing

- Entry: through a POST event and by means of CreateStationAction, UploadDataAction, and RegisterStationAction.
- Exit: exit event is only an confirmation message.
- Read: no read, as this process is a writing.
- Write: through INSERT queries on the database, and Logger methods that perform log recording.

database reconstruction

- Entry: no entry, as it reconstruction is activated through a request, at server side, that does not cross the boundary.
- Exit: no exit, as before.
- Read: log reading
- Write: through a reconstruction method.

Size of functional processes.
For every identified functional process, we estimate the weight of a process by counting the number of events that are related to it.

- Size(log recording) = 3 entry + 1 write = 4
- Size(database reading) = 3 entry + 1 exit + 1 read = 5
- Size(database writing) = 3 entry + 1 exit + 1 write = 5
- Size(database reconstruction) = 1 read + 1 write = 2

Total values.
We sum the contribute of every functional process, for every COSMIC action. Entry=9, Read=2, Write=3, Exit=2

11.2 | MVC version.

The analysis of all the aspects up to the Triggering events is the same, and therefore they are omitted.

Data groups.
For every functional process, the corresponding COSMIC actions are identified.
log recording

- Entry: no entry, as MVC does not enable native log recording.
- Exit: no exit, as above.
- Read: no read, as above.
- Write: no write, as above.

database reading

- Entry: through GET events that are equivalent to DownloadDataAction, LoadMinimizedStationsAction, and QueryDataGraphAction in oMVC.
- Exit: it is a cvs file including selected data. It occurs through an HTTPResponse.
- Read: it is achieved through SELECT queries in the database.
- Write: no write, as this process is a reading.

database writing

- Entry: through POST events that are equivalent to CreateStationAction, UploadDataAction, and RegisterStationAction in oMVC.
- Exit: exit event is only an confirmation message.
- Read: no read, as this process is a writing.
- Write: through INSERT queries on the database, and Logger methods that perform log recording.

database reconstruction

- Entry: no entry, as MVC does not enable native database reconstruction.
- Exit: no exit, as above.
- Read: no read, as above.
- Write: no write, as above.

Size of functional processes.
For every identified functional process, we estimate the weight of a process by counting the number of events that are related to it.

- Size(log recording) = 0
- Size(database reading) = 3 entry + 1 exit + 1 read = 5
- Size(database writing) = 3 entry + 1 exit + 1 write = 5
- Size(database reconstruction) = 0

Total values.
We sum the contribute of every functional process, for every COSMIC action. Entry=6, Read=1, Write=1, Exit=2

COSMIC FP	Observable	Unobservable	Var.
Entry	9	6	+50%
Exit	2	2	+0%
Read	2	1	+100%
Write	3	1	+150%

TABLE 7 COSMIC Function Point counts for observable (col. 2) and unobservable (col. 3) architectures, with evaluation of variance of observable with respect to unobservable architectures (col. 4) across the types of data movement counts in the COSMIC taxonomy (col. 1).