MASTER IN COMPUTER SCIENCE

Artio: Large Scale Investigation of Software Variability in Nostr

Master Thesis

Michael Kaiser

Philosophisch-naturwissenschaftlichen Fakultät der Universitat Bern

Prof. Dr. Timo Kehrer
Roman Bögli

Software Engineering Group
Universität Bern

May 2026

UNIVERSITÄT BERN

UNIVERSITÉ DE NEUCHÂTEL

UNIVERSITÉ DE FRIBOURG
UNIVERSITÄT FREIBURG

Abstract

The recently emerging terminology of *Community-Driven Variability* (CDV) [1, 2] describes a new kind of software variability paradigm that is different from classical, variability-intensive systems such as Software Product Line (SPL) engineering. Different implementations of the same functionality can emerge from the community without a central authority, which presents both opportunities and challenges for software development and research.

The manifestation of CDV can be observed in the Nostr protocol [3], which is a decentralized communication protocol built with relays, clients, and events (messages) routed among them. As common for CDV-exhibiting ecosystems, the feature space of Nostr-compatible software applications is shaped by crowdsourced specification documents known as *Nostr Implementation Possibilities* (NIPs). Within this thesis we introduce *Artio*, a platform for investigating how the Nostr ecosystem is used and how its variability dimensions manifest in the wild. Guided by three main research questions, we examine (RQ1) the diversity of relay applications and (RQ2) events, the (RQ3) NIP-conformity of messages, and (RQ4) overall usage trends.

Our results indicate a rapidly growing ecosystem with a diverse set of software implementations, relays, and users. We further observe that specification violations are common rather than exceptional, with discernible patterns in co-occurring NIPs and their likelihood of violation. All insights are enabled by our Artio research platform, an additional contribution of this thesis, which can be used for further research on the Nostr ecosystem and related systems.

Keywords: Community-Driven Variability, Nostr Protocol, Software Variability Analysis, Implementation Possibilities, Software Ecosystem, Decentralized Systems

Contents

1 Introduction

Over the past years, the software landscape has witnessed a significant shift in the way software is developed, maintained, and evolved. With the rise of open-source communities, decentralized development models, and the increase in challenges for software variability management, there is a growing interest in understanding the implications and results that emerge from these new paradigms and trends of software development. This thesis presents a comprehensive investigation into the field of *Community-Driven Variability* (CDV) [1, 2] by means of the Nostr [3] software ecosystem as case study. Earlier studies [2, 4] have already investigated the paradigm of CDV and made comparisons between different software ecosystems that exhibit CDV. However, these studies have been more focused on providing an overview and understanding of the general phenomenon of CDV across different ecosystems. While the contributions provided valuable insights and the foundational work necessary for further research on CDV they have not made a detailed investigation of the manifestation of CDV in one specific ecosystem and the implications of it. This is where this thesis takes a more in-depth and focused approach by investigating a single ecosystem in more detail with the introduction of a platform that enables the investigation of large scale software problems that require data collection and analysis from multiple sources. The proposed platform is used to investigate the Nostr protocol and its ecosystem as introduced in Chapter 3 leading to a better understanding of the specific challenges and implications of CDV in this ecosystem and providing insights that can be applied to other ecosystems as well.

1.1 Motivation and Contribution

In nowadays software development world, variability and its management is a crucial aspect of software engineering. Given more classical approaches on managing variability in software systems, such as *software product-line* (SPL) engineering [5, 6] that rely on centralized processes and controls, there is a growing interest in understanding the implications and results that emerges from decentralized and "community driven variability". One example of such community driven variability can be found in the Nostr protocol [3] and its software ecosystem around it, where different implementations and usage patterns emerge from the community without a central authority controlling or managing it. The main

protocol is defined and developed by the community but the implementations and usage of the protocol can vary widely opening up a wide range of research questions and challenges. The thesis provides insight into this ecosystem and find out if there are any patterns, trends or lessons to be learned from the decentralized approach to software development.

The main contributions of this thesis are:

1. **Theoretical Contribution**: An in-depth analysis of the phenomenon of community-driven variability in software ecosystems, with a specific focus on the Nostr protocol and its ecosystem as a case study leading to a better understanding of the specific challenges and implications of CDV in this ecosystem and providing insights that can be applied to other ecosystems as well.
2. **Methodological Contribution**: Adaption of existing methods and structures into a multi-component platform for analyzing large software systems for variability and long-term observations as described in Chapter 3.
3. **Empirical Contribution**: Comprehensive evaluation demonstrating the effectiveness of the proposed approach in the real-world case of Nostr [3] guided by 4 research questions that are answered in Chapter 4.
4. **Practical Contribution**: Implementation and usage of a working system that can be applied in real-world scenarios including discussion of our approach. This includes the implementation of the go-db-etl [7] replication package that is used to collect and historize data from multiple sources in a single database.

Thesis Organization This thesis is organized as follows:

- **Chapter 1** provides an introduction to the topic and contributions.
- **Chapter 2** presents a comprehensive literature review and theoretical background.
- **Chapter 3** describes the specific research questions and objectives including the implementation and experimental setup.
- **Chapter 4** presents the results and evaluation of the proposed approach.
- **Chapter 5** discusses the implications, limitations, and future work.
- **Chapter 6** concludes the thesis and summarizes the main findings.

1.2 Related Work

Previous work on the topic of large-scale investigation of Nostr has been conducted by [8], where the authors have conducted an empirical analysis of the Nostr protocol focusing on relays and replication strategies and their post-replication accross relays. Besides this work,other researchers have already conducted research on the topic of CDV and Online Social Networks. While [9] mainly focused on the architectural challenges and the comparison between different approaches for decentralized online social networks. The work of [10] is more focused on attack vectors and security implications of Nostr, with Proof-of-Concept and mitigations for the identified attack vectors. Our work differentiates itself from these works by providing a more comprehensive and in-depth analysis of the Nostr ecosystem, with a focus on the implications of CDV and the development of a platform for analyzing large software systems for variability and long-term observations.

1.3 Summary

This chapter has introduced the research problem and the general organization of the thesis. The following chapters will provide detailed discussions of the background, proposed methods, experimental results, and conclusions. The work presented here contributes to advancing the state of knowledge in community driven software variability and provides practical solutions for analyzing and understanding the Nostr protocol ecosystem or other related systems.

2 Background

First of all, we will introduce the main concepts and background that are needed to understand the proceeding work and the research questions that we will be investigating in this thesis. These explanations will help to understand the context of the research conducted in this thesis and the motivation behind it.

2.1 Community-Driven Variability

Recent research has highlighted a new kind of software variability paradigm termed *community-driven variability* (CDV) [1, 2]. This kind of software variability is different from conventional, well-known software variability paradigms such as software product lines (SPLs) [6, 11, 12]. The difference in paradigm emerges in systems where software components or their protocols are developed and maintained by an open community of users rather than a centralized authority or selected limited group of maintainers. The focus of CDV lies on achieving interoperability within the software through an extensible protocol design. This design is achieved through so-called *Implementation Possibilities* (IPs) which are specification documents for specific functionalities (features). The previous work [2] has found such CDV to be a central symptom in decentralized systems such as Nostr. Since ecosystems such as Nostr are still developing and evolving, the CDV paradigm is still in its early stages and there are many open questions regarding how it works in practice, how it can be enforced, and how it can be handled when there are protocol violations. Coming from these open questions, it is of interest to investigate the CDV phenomenon in practice and see how they are manifested in the wild and how they could influence or enhance the traditional and better-known software variability paradigms such as SPLs.

2.2 Nostr

Nostr is the abbreviation for *"Notes and Other Stuff Transmitted by Relays"* and is a decentralized protocol for creating a global, censorship-resistant social network. It was created by a user named fiatjaf in October 2022 [3]. The main idea behind Nostr is to provide an open protocol that allows users to publish and share content without relying on centralized servers or platforms.

Through the IPs found in CDV-exhibiting software systems, which are called NIPs (Nostr Implementation Possibilities) [13] in the Nostr ecosystem, developers from the community can derive their own software with different features and capabilities while still being interoperable with other Nostr protocol, given that they respect and implement the core (mandatory) NIPs correctly. The full set of NIPs is publicly available at the nostr-protocol github repository [13], where everyone can contribute according to the information found in the README file. As of 2026, there exist over 100 NIPs, of which only the first one is mandatory to be implemented. All other NIPs are optional and can be implemented by the developers of the relays or clients as they wish, which unfolds additional dimensions of variability in software. Such dimensions can be unfolded in different ways, such as the fact that one feature might enable the usage of entirely new protocols based upon this feature. Some main ideas of such functionalities and features will later be highlighted in the Chapter 5.

The ecosystem of Nostr consists of two main components: *clients* and *relays*. The communication between these two components is how they interact and how the users can publish and consume content on the Nostr network. The message exchange with its specific JSON-based structure is part of the mandatory `NIP01` and is the basis for all communication in the Nostr ecosystem, which is referred to as the distribution of messages between clients and relays. Clients are applications where the user directly interacts with the Nostr network through a graphical user interface (GUI). While there exist CLI clients as well, they are not the ones that are promoted by the Nostr community and their websites [3, 14]. A popular example of such a client is *Damus* [15], which is available on iOS and Android. Other clients may be web-based, such as *Primal* [16]. Relays, on the other hand, are servers that store and distribute the content published by the users and their clients. The relays are responsible for receiving, storing, and distributing the messages published by the users. One basic implementation of a relay that implements basic NIPs can be found at the github repository of nostr-relayer [17]. There exist many other variations of apps that interface with the Nostr protocol, such as bots, media sharing and other tools as displayed at https://nostrapps.com/.

The main communication between clients and relays is done through WebSockets, where the client and relays can exchange messages over an established connection. The messages themselves are JSON objects that are transmitted in UTF-8 byte sequences, where for both sides of the communication to make sense of the messages. Given these streams they are then able to parse the byte stream into a correct JSON object and continue to interpret the content of the message. This interpretation is guided by the NIPs that are implemented on the relay and client side. For most of the messages, there is only `NIP01` that defines the functionality, but for other NIPs, such as `NIP45` described in Section 2.2, there are specific message types that are defined in the NIP itself. In this example, it is the *COUNT* message type that is newly introduced in order to allow clients to query the relays for the amount of events that match a given filter. A more thorough introduction and explanation of the event types will be given in Chapter 4 when we analyze the event landscape in more detail.

The implementation of the NIPs is not only left to the clients but also to the relays. The amount of changes required to implement a NIP can vary greatly in general but also between client and relay implementations. Both clients and relays do not have any restrictions on the concrete implementation in any form, as long as they implement the protocol as defined in the NIPs that the developers choose to implement, given they want to achieve interoperability with other clients and relays that implement the same NIPs. However, the Nostr protocol is indicatively more restrictive for the relays since the clients rely on the correctness of the relays to function properly. If one wants to make a relay that should achieve a high usage rate in the network, then one needs to implement the NIPs correctly, since users will not use a relay that does not demonstrate proper interoperability with their chosen client if working relays are available. This liberty given to the developers and users of Nostr raises the question of how and if such protocols can be and are enforced, if there are protocol violations visible in the wild, and how such violations can be handled. In order to provide a foundational understanding of the NIP space that is needed for the proceeding work, we

will highlight some of the NIPs in the following.

NIP-01 NIP01 defines the core protocol of Nostr and is the only mandatory NIP [18]. It states that "*[t]his NIP defines the basic protocol that should be implemented by everybody*" [18]. It defines the core objects and structures, such as events, signatures, tags, and the communication between clients and relays. The most important part of NIP-01 is the definition of the main event structure, i.e., how to compile correct messages. Within it, the main message field is contained and titled as the content. Many other NIPs build on top of this content field, such as NIP65.

```
{
  'id': '3ccfcc19109463cee70d34b5456884a93b9389958182dcba75feb20c26463dc7',
  'pubkey': '7719cbfe13490585229c37b7677f9db95ad58bd5dcdeb01b312a53faaebfeaca',
  'created_at': 1769171882,
  'kind': 20000,
  'tags': [
    ['g', 'dr']
  ],
  'content': 'hello there!',
  'sig': 'fb7ca30b94e643d35bffc4aabc42f0404e67d1678f84117fc6f7700a782d4ef53618015c54ba8d19d0f0e3989a75472827aa2966211bf42b55f3dc42d89d8cef'
}
```

Listing 2.1: NIP-01 example event found on relay.artiostr.ch

NIP-11 NIP11, titled "Relay Information Document" [19], defines a standard way for relays to publish information about themselves, such as their operator's public key, software stack, version, and supported NIPs. This NIP needs to be implemented by the relays so that clients can then consume the provided information and use it for various purposes if they see fit. An example of such a relay information document as implemented on our own relays can be found in Listing 2.3. Using a simple `curl` command with the correct header (as displayed in Listing 2.2), one can retrieve this information from the relay. The result of this command is the content of Listing 2.3, which is a JSON object containing the information about the relay, such as its name, description, public key, contact information, supported NIPs, software used, version, icon, and banner.

```
curl --location 'http://relay.artio.inf.unibe.ch/' --header 'Accept: application/nostr+json'
```

Listing 2.2: NIP-11 Retrieval Using Curl From relay.artio.inf.unibe.ch

NIP-45 NIP45, titled "Event Counts", introduces a new event type called COUNT, which is used to query relays for the amount of events that match a given filter. This NIP introduces new functionality with which we can query and count the amount of messages for a given filter (e.g. events a user has published) on a relay without having to transmit all the events from the relay to the client and then count them on the client side.

NIP-65 NIP65, titled "Relay List Metadata", introduces another new event type that is used to publish a list of relays from a user. This list indicates which relays a user is using in a read or write manner. It is necessary to see where one can find events published by a user or where one can publish an event in the form of a direct message to the given user.

```
{
    'name': '',
    'description': 'artio relayer implementation for the university of bern',
    'pubkey': 'b28ba0033d6f35d54c6955d900ac191f6e7bcefa71bcf67874e6ca267184580d',
    'contact': 'https://seg.inf.unibe.ch/contact/',
    'supported_nips': [
        1,
        2,
        9,
        11,
        45,
        50,
        65,
        42
    ],
    'software': 'https://github.com/SEG-UNIBE/artio-relay',
    'version': '1.1',
    'icon': 'https://raw.githubusercontent.com/SEG-UNIBE/artio-relay/main/identity/logo_relay.svg'
    ,
    'banner': 'https://raw.githubusercontent.com/SEG-UNIBE/artio-relay/main/identity/logo_relay.
    svg'
}
```

Listing 2.3: NIP-11 Response relay.artio.inf.unibe.ch

The CDV paradigm found in Nostr is clearly visible in the main repository of Nostr [13], such as backwards-compatibility and the requirement that NIPs should be fully implemented in at least two clients and one relay before being considered for acceptance into the NIP catalog. The openness and community-ownership of the Nostr protocol is highlighted several times. This includes the note that when repository owners (who are a technical necessity that cannot be avoided when using GitHub as a platform) act in an unjustified or malicious way, the community must react, indicating that social consensus can be more powerful than access rights to the protocol repository. The 738 forks of the repository (as of March 2026) guarantee that a potential history rewrite from a repository owner can be easily mitigated by the community.

2.3 Building on Prior Work

In the realm of ongoing CDV research [1, 2, 4], the Software Engineering Group at the University of Bern [20] started to investigate Nostr. A first set of goals has been defined as follows:

- Understand the Nostr protocol and its ecosystem.
- Develop a Nostr *watchtower*.
- Get first insights into the spectrum and finesse of Nostr.

The wording of *watchtower* has been defined loosely in order to allow for creativity and freedom. It also allowed the researchers to not be restricted on how to build such a watchtower and to adapt to new findings throughout the research process. The outcome of the seminar has been named *Artio*, inspired by a Celtic goddess associated with wildlife and hunting.

Artio [21] is the result of this research process and the main subject of this thesis. A first version has been developed and published in the context of previous work [22]. It proposes a platform that allows for collecting, storing, and analyzing data from the Nostr network and then analyzing it in a meaningful way that emerged throughout the proceeding work leading to this thesis. The first version contained the

artio-relay [23] in a first iteration and artio-insight as a first idea on how to collect data. The first version has then been further developed throughout this thesis to allow for more complex data collection and analysis. The main research questions are defined in Chapter 3 and then answered through the implementation described in Chapter 3 and the results presented in Chapter 4.

3 Methodology

Given the previous introduction to the Nostr ecosystem and the conducted groundwork research over the CDV paradigm, we are now ready to dive into the actual research questions and the methodology we will use to answer them.

3.1 Research Questions

Given the already existing implementation of Artio as described in Section 2.3, we define four research questions to guide the CDV-related investigation of the Nostr ecosystem. The research questions that we will later further divide into sub-questions for this thesis are:

RQ1: Diversity of Relay Landscape The background for RQ1 is based on the fact that the implementation of the Nostr protocol is sometimes available in GitHub repositories, but the adoption rate of such implementations is not known and the amount of non-public implementations is impossible to detect just looking at public code bases. The NIP-11 documents specify the software stack and more information about the relay and are the only technical source of truth. This truth is, as it is the spirit of the protocol, decentralized and thus not accessible. To achieve a good understanding of the relay landscape one cannot only statically analyze code bases, but must get an actual understanding of the real world usage to grasp possible issues and trends that might influence the research. In this first RQ, we therefore poste the following sub-questions:

RQ1: How diverse is the *relay landscape* based on ...

RQ1.1: Unique operator pubkeys (NIP-11)

RQ1.2: Software stack and version (NIP-11)

RQ1.3: Supported NIPs and their combinations (NIP-11)

RQ2: Diversity of Event Landscape While RQ1 is focused on the relay landscape, RQ2 is focused on the event landscape. The motivation is to see what kind of messages and interactions are actually being transmitted by users to relays within the network and find possible trends and patterns that might be a direct result of the CDV paradigm. Apart from the paradigm, also a social view into the protocol adoption is possible by looking what kinds of events are actually used and in what amounts, highlighting the most used features. This shows the actual projection and implementation result of the protocol and its features in the wild and not just in the code bases, manifesting the real world usage and adoption of the protocol. The emerging research questions are the following:

RQ2: How diverse is the *event landscape* based on ...

RQ2.1: Message types (NIP-01)

RQ2.2: Message kinds (cf. nostr.dev [24])

RQ3: NIP Conformance of Messages Since the impact of CDV on any given software ecosystem is not yet fully understood, it is currently not possible to give a clear answer whether the statement about backwards compatibility in the protocol [13] is actually true in the real world. Whenever the question of backward compatibility arises, the complementary question of forward compatibility follows, i.e., whether existing implementations must adopt new features to maintain interoperability. For example, does a Nostr relay require code changes to support a new NIP? This is interesting to see, since when complete forwards compatibility is found and proven functional, the impact of CDV on current development processes might be quite high. Even if there is no full backwards or forwards compatibility, the question about the extent of compatibility is still interesting since it might generate some interesting and unexpected findings about compatibility between software version, that can be adopted into today's more traditional variability management approaches such as, e.g., SPL engineering [6, 11, 12]. Considering today's decentralized software architecture approaches (such as microservices, etc.) that are also event based and have to deal with similar issues of variability and compatibility, the findings from this research question might be applicable to those approaches as well.

RQ3: How NIP-conform are messages propagated in the network?

RQ3.1: How many officially non-supported-kind messages did we receive?

RQ3.2: To what extent do relays fail to handle message kinds of allegedly supported NIPs?

RQ4: Usage Trends in the ecosystem Lastly, RQ4 focuses on interactions of users and relays with each other or themselves. The motivation is to see if there are any patterns in the interactions between users and relays that might be a direct result of the CDV paradigm. The previous research questions might also be influenced by the interactions from users with clients and vice-versa. For example, it is possible for a relay to not be detected from certain users and relays due to the fact that they are only used by a small amount of users and thus not advertised in the NIP-65 events. As stated in the introduction Chapter 1, the liberty of NIP selection and implementation unfolds new dimensions of variability, which might be able to be observed and analyzed through interactions and usage patterns.

RQ4: How do usage trends manifest in the Nostr ecosystem in ...

RQ4.1: Relay Discovery by Software

RQ4.2: User to Relay Usage Patterns

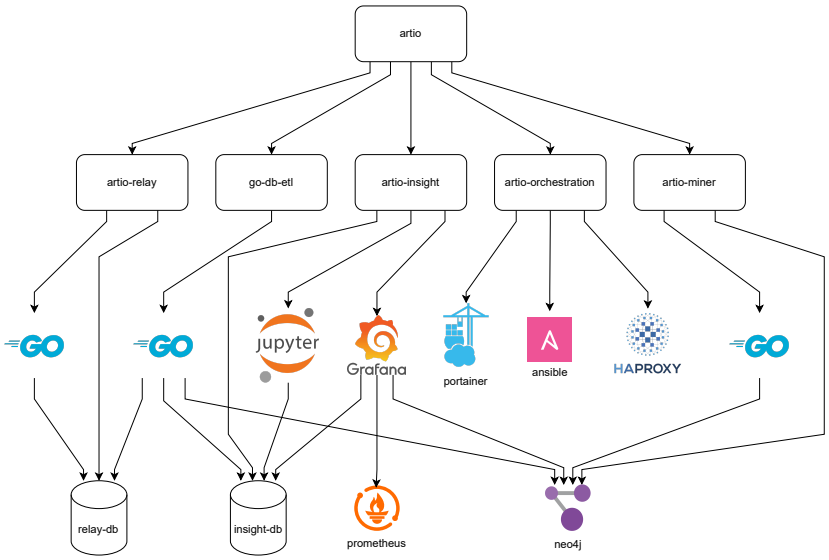


Figure 3.1: Artio Overview

3.2 Components

In order to answers these research questions defined in Section 3.1, we had to implement a platform capable of collecting, storing and analyzing data from the live Nostr network. This chapter describes the architecture of the overarching platform called Artio [21], its components and the interactions between these components. A high-level view of the architecture is visible in Figure 3.1, where the components and their given dependencies are shown. From the top to the bottom we can see four layers where the first is the full platform, that is then split up for the second layer. This layer contains the five components which will be described in detail in the following sections. The third layer contains the technologies for each of the components and is linked to the fourth layer representing the data storages for each of the components including their dependencies.

3.2.1 artio-relay

The artio-relay [23] is the relay component of Artio and is the first component that has been deployed. It is actively participating in the Nostr network as a relay and is open to use for any users. It is based on the simple implementation found at [17] and has been modified to fit the needs of Artio. The first implementation of the artio-relay, in the scope of the preliminary work [22], was minimal and only focused on the core relay functionality, but kept in mind that in the future an extension to the relay might be necessary. During the investigation process the relay has been extended with functionality defined in the NIPs [13] and also with custom functionality to allow for better data collection and observability. Such functionality includes a custom prometheus exporter [23, 25, 26], that we later use to collect metrics about the relay and its performance (message received, messages sent, etc.). These exporters and other custom functionality such as logging are not required, nor defined in any NIP, but are used to gain insights into the usage and performance of our relays. During all this process, we also refactored the code base to be more modular and human readable, to allow for easier extensions and maintenance.

The artio-relay is written in Go [27], uses a PostgreSQL [28] database storage backend and is containerized using Docker [29]. For our purposes we have two instances of the artio-relay running, one at the University of Bern[1] and one in a private Homelab[2]. The artiostr relay is mainly used for testing the software and for development purposes, while the unibe relay is considered the production relay. However, the artiostr relay has been up and running for a longer period of time dating back to April 2025 and thus has seen more traffic and messages than the unibe relay. Since the data from the artiostr relay is not any less valuable we are also including this data in this study. Both relays are actively participating in the Nostr network and are used for data collection and analysis.

3.2.2 artio-miner

The artio-miner is a custom component developed to mine the Nostr ecosystem for data and represents a standalone component [30]. It is written in Go [27] and is responsible for actively fetching data from the Nostr network using our two relays as a starting point. The data fetched from the artio-miner is highly relational since it is based on the interactions between users and relays by means of transmitted messages. To store the collected data, we use the graph database Neo4j [31], which enables native querying of network-structured data while preserving flexibility to add new fields and relationships.

We leverage certain widely-adopted NIPs of the Nostr protocol to gain insights into the network. Specifically, these NIPs are:

- `NIP11` the *relay information document* as introduced in Section 2.2 [19].
- `NIP65` the *relay list metadata* as introduced in Section 2.2 [32].

Using these two NIPs and data collected from our relays, we can infer properties of other relays and the broader network topology. This data is directly coming from the NIP-65 events that users are publishing to our relays. Such events are obtained by using a request message that filters for kind 10002 events, i.e., `NIP65` events. By analyzing these events, we are capable of building a list of relays that are advertised to be in use by the users and their clients. Using this list, we iterate over the newly discovered relays to obtain more information from those relays in the same way we did with our own relays. By running this process for a longer period of time we are able to build a large dataset of relays and their information, as well as the users that are using those relays. While doing so we are also collecting the NIP-11 documents from those relays to learn about software stack usages and what versions they are running. An example of such a NIP-11 document is shown in Listing 2.3.

From an implementation point of view the artio-miner is designed to start by fetching the NIP-11 documents from our two relays and then start the process of extracting new relays from these events and then continue to fetch the NIP-11 documents from the newly discovered relays while also keeping track of the already discovered relays to avoid loops and unnecessary requests. Using the builtin Go functionalities, we implemented this process in a concurrent way, allowing us to fetch data from multiple relays at the same time and thus speed up the data collection process. The scaling of the fetching processing and how many workers we are running is purely dependent on the system performance from the server we are running the artio-miner on.

For further analysis we are able to export the data from the graph database into a CSV file for further processing.

[1] relay.artio.inf.unibe.ch
[2] relay.artiostr.ch

3.2.3 go-db-etl

Given the postgres database used by the artio-relay [23], and the neo4j database of the artio-miner [30] we are not yet able to do any analysis on these datasets in a centralized way. A central database will be introduced later in the context of the artio-insight (cf. Section 3.2.4) component. However, we lack a way to obtain the data from the source databases and load it into the target database. For this specific reason a custom ETL (Extract, Transform, Load) tool has been developed using the Go programming language [27]. The go-db-etl tool [7] is responsible for extracting data from several sources, transforming it into PostgreSQL compatible table formats and loading it into the target database. Within the target database a history of the data is built. This history allows us to gain insights into data, that is no longer available in the source systems, such as deleted events on the relay or deleted/rotated log messages.

The approach underlying this ETL tool is derived from the principles about data warehousing described by Linstedt and Olschimke [33] and adapted into the context of our application as visible in Figure 3.2. However, ETL is not the appropriate term here, as our process follows an ELT (Extract, Load, Transform) approach, where transformations are performed in the target database using SQL queries. This allows for easier adjustments of the transform logic since it is done in one location and we can minimize the computational impact on the source systems we are loading. In the artio-relay context performance impacts on the source systems could be accepted, since the source systems are not business critical and generating a revenue stream. The need for low to no performance impact on the source systems would be a high priority requirement in business critical systems, where the data is generated by a revenue stream and any performance impact could lead to a loss of revenue.

In our case, however, we are also interested in investigating data that is no longer available in the source system or has been modified. This leads to the need of historization of the data, to allow us to track and spot changes that are not possible by just looking at the source systems. The decision on if such a change or deletion is of interest cannot be made in a generic way and needs to be made on a case-by-case basis. Without the historization of the data we would not even be capable of recognizing such changes.

For the historization we are using different database schemas.

- **INB (Inbound):** This schema contains the raw data as it is extracted from the source systems with no processing. It uses unlogged tables in order to achieve the best possible performance for the loading process over the network.
- **RDV (Raw Data Vault):** This schema contains the data from the INB schema, but is enriched with additional metadata such as the timestamp of the load, the source system and other relevant information. This layer also historizes the data by tracking records no longer present in the source systems.
- **`UDM` (Unified Data model):** This schema contains the data from the RDV schema, but is transformed into the analytical data model as defined in the artio-insight (cf. Section 3.2.4) component.

The segmentation of these three schemas allows us to have a clear separation of responsibilities. The INB schema allows us to achieve the goal of reducing the performance impact on the source systems, while the RDV schema allows us to achieve the goal of data historization. The `UDM` schema is not part of the go-db-etl component but belongs to the artio-insight component, which is discussed in detail in Section 3.2.4.

Example An example of the benefit of the historization of the data is the case of the log table from the artio-relay [23]. Over time, several million rows have accumulated in this table on our relays, leading to the database running out of storage space. Having the data historized in the RDV schema allows us to delete the data in the source system, while still having all data available in the target database for retrospective analysis and visualization. For normal data points (normal being that the person conducting the data

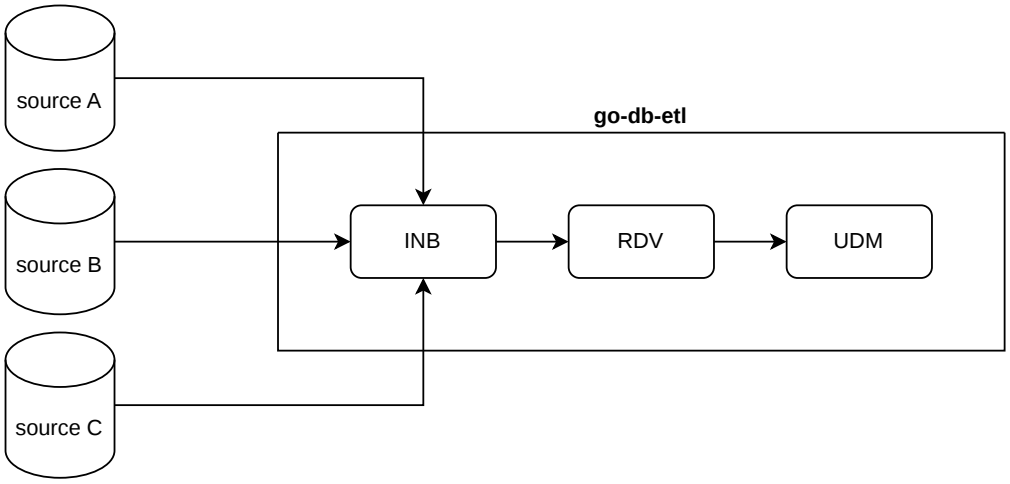


Figure 3.2: go-db-etl Flow

analysis not having full control over the source system, contrary to our position with the artio-relay [23]) this is not possible, however since we have the full knowledge on the target and rdv system we can delete the data in the source system without experiencing any problems within the application due to the fully transactional nature of the log table.

3.2.4 artio-insight

As mentioned in the previous section for the go-db-etl component (cf. Section 3.2.3), the artio-insight [34] component of Artio is responsible for storing and processing data from multiple sources in a historized manner for analysis and visualization to reach the goal of answering the research questions defined in Section 3.1. To speak of it as a single component is misleading, since it is composed of multiple sub-components. The components with their respective responsibilities are:

- **Data Analytics:** Responsible for storing and providing data for analysis. This part is achieved using PostgreSQL [28] as database and neo4j [31] as a graph database.
- **Data Visualization:** Responsible for visualizing the collected data in a meaningful way. This part is achieved using Grafana [35] and Jupyter Notebooks. The output of these components can be seen in Chapter 4.
- **System Monitoring:** Responsible for monitoring the health and performance of the overall system. This part is achieved using Prometheus [25].

The usage of these components (especially Grafana [35], Prometheus [25] and a self-defined PostgreSQL [28] database) allows for a quick adoption for new metrics and data points. The most important sub-component of artio-insight is the `UDM` schema that defines the analytical data model. It can be seen in Figure 3.3.

A short preview of the `UDM` schema as visible in Figure 3.3 lets us make educated guesses that the key tables are gonna be the *udm.relay* table, that contains all the information about the relays in the network, and the *udm.event* table, that contains all the events received by our relays. The *udm.relay* table is gonna

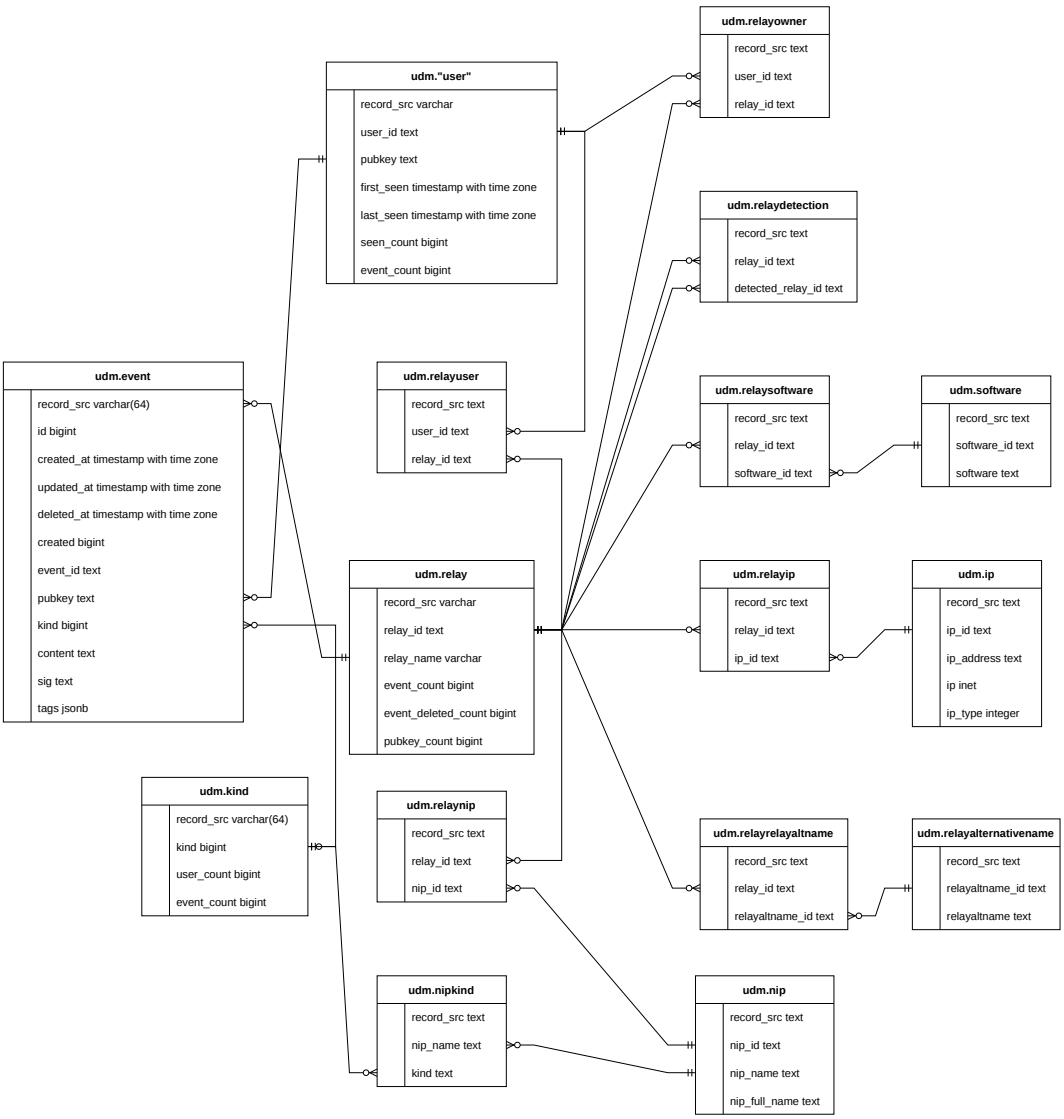


Figure 3.3: UDM Data Model Overview

be the main source for answering lots of questions since they act as our sources of information for the thesis, but also for the whole nostr ecosystem.

The data from the UDM is pursued for all the analysis and visualizations done in this thesis and is the main data source for answering the research questions defined in Chapter 3. The artio-insight component is not responsible for the active data collection, but rather for the passive, long-term data storage. The active data collection is done by the artio-relay (cf. Section 3.2.1) and the artio-miner (cf. Section 3.2.2). This decoupling from the source systems allows us to have a more flexible and scalable architecture, while not being dependent on the performance or implementation details of the source systems. The visualizations and analysis are programmed using a Jupyter Notebooks. The benefit of this solution is that we can use plain SQL queries to obtain the data we need and load into the notebook. From there we can then transform and aggregate the data using Python and visualize it using libraries such as Matplotlib [36] and Seaborn [37]. The advantage of this solution is its reproducibility, since the entire analysis and visualization process is present in the notebook and can be rerun with new data or after changes in the data model. Additional Python code automatically exports visualizations as PDFs, SVGs and generates tables for use in the presented results in Chapter 4. The data from the prometheus monitoring is not included in the UDM as its main purpose is to monitor the health and performance of the system and is not actively used for our analysis. This is the reason that in Figure 3.1 the prometheus component is not connected to the UDM.

The load scripts for the UDM are included with the GitHub repository of the component, but the deployment of the scripts is not automated. We decided not to automate the deployment due to several reasons, that we will showcase in Section 3.2.5. The load scripts are designed not as plain SQL scripts but rather as stored procedures in the database, that can be called with a boolean force parameter. This parameter decides if we will only truncate and reload the data, or if we will also drop and recreate the tables with their corresponding indexes. In order to allow for a more flexible analysis process, we did not implement any foreign keys in the UDM, since they would make the loading process more complex and would also restrict some of the analysis where we rely on relations between data points not being present. An overview of the whole data flow from start to end will be given in Section 3.2.6.

3.2.5 artio-orchestration

As the previously described components show, the overall system cannot be provisioned with a single docker-compose command. For the setup and orchestration of the overall Artio platform we use Ansible [38] as our automation tool of choice. The artio-orchestration [39] repository contains all the necessary playbooks, roles and inventory files to deploy and manage our test and productive server instances artiostr and unibe, respectively. Furthermore it contains also the necessary configuration files for all the components such as Grafana, Prometheus and PostgreSQL. For each of these components there exists at least one ansible role that is responsible for the component's setup and configuration. An example of such a role is the artio-relay role. It will install the necessary dependencies, such as git, docker and docker-compose, copy over the env file containing the necessary variables such as the passwords and secret keys. For the main application it will clone the repository and then run the docker compose file to get the application up and running. One benefit of using ansible is the fact that we can redeploy Artio or components of it in case of a failure or renewing versions of the components. It gives us the trust in the system that we can recover from failures and ensure that the system is setup in a consistent way across all the components and not dependent on manual steps that might get forgotten.

The whole setup for the production environment can be triggered using the command shown in Listing 3.1, which will run the `site.yml` playbook with the inventory file for the production environment. Yet, some manual work remains for the initial setup of the servers, such as setting up the firewall rules and port forwarding, and the database configuration for the artio-insight component. Due to the fact that it contains

```
ansible-playbook site.yml -i inventory/production/hosts.yml -l artio.inf.unibe.ch
```

Listing 3.1: Ansible setup command

sensitive information such as passwords and secret keys, the artio-orchestration repository remains private.

3.2.6 Complete Data Flow

Now combining all the components described in the previous section, we can describe the complete data flow of the Artio platfrom. The data flow starts at the artio-relay component that is actively participating in the Nostr network as a relay, together with the data from our miner and other files. These are then fetched from the go-db-etl component and loaded and historized into the artio-insight component in the RDV schema. Given that we previously created the stored procedures for the `UDM`, the load procedure is then used to transform the data from the RDV schema into the `UDM` schema. The full data flow can be seen in Section A.1.

Once the setup has been finished with the orchestration component, artio is up and running and ready to start the data flow for analysis. The data from our two relays and the miner is extracted and loaded into the artio-insight component using the go-db-etl component. Up to the RDV schema, the process is completely automated and does not require any manual steps. From the RDV we can then use SQL queries in the form of procedures to transform the data into a more analysis and normalized form ready for analysis and visualization. The queries are not automated since they are not generic. Analysis of the data requires intricate knowledge about the domain and the research questions, since they dictate the needed forms and structures. An example for the `udm.kind` table is shown in Listing A.2 in the appendix, where we create the procedure for loading and/or creating the tables. Given the existence of this procedure, the full data load can be triggered using one singular stored procedure, the `udm.run_all_loads`, that will call all the individual load procedures for the `UDM` schema (cf. Listing A.1).

At this point we can access our data using the described Grafana dashboards and Jupyter Notebooks for analysis and visualization, and can start to answer the research questions defined in Section 3.1.

3.3 Deployment

The described components are static and not doing any work. We are required to deploy the components with the artio-orchestration (cf. Section 3.2.5) component to get the system up and running. The deployment is realized with two servers. The first server is located in the homelab of Michael Kaiser and was first used as the initial relay in the groundwork phase of the research. It is still up, running and acting as a relay being available under the hostname *relay.artiostr.ch*. The access to the Nostr relay is realized through a reverse proxy using Cloudflare [40] while the database is running on the same server and is only accessible from the local network. This guarantees that the relay is available to the public, while the database is not accessible to the public and is only used for development and testing purposes.

For a proper research setup we required a more stable solution that is not dependent on the availability of a single person but rather on the availability of a university infrastructure. This led to the introduction of the server at the University of Bern, which is available under the hostname *relay.artio.inf.unibe.ch*. It is now acting as the production environment for the research and is made publicly available through direct access via the IP address. Access to the database and other non-public services is only possible within the university network and is protected with a firewall. HTTP/HTTPS access is made possible through a firewall rule setup from the university.

Using the go-db-etl component we are extracting the data from both relays and loading it into the artio-insight component for analysis. The historization aspect of go-db-etl allows us to track changes in the data and for the case that the first private relay would become unavailable, we are capable of using the historized data to continue the research. The database of the artiostr relay is made available to the artioinf server through port forwarding. This introduces a risk since the database is now publicly available, but we handle this risk by using an IP whitelist only allowing the artioinf server to access the database. Further security measures include daily and weekly backups of the whole artiostr server and its data.

If any problems arise with the applications themselves but the underlying data storage is not affected, we can leverage the advantages of the orchestration component to quickly redeploy the application and get them up and running again. The exact same applies for the case of any updates to the components, that do not require any changes to be made to the persistent data, which are not handled by the components themselves nor the orchestration component.

Given these explanations about the deployment and the data flow, a basic understanding of the data and its structure is now available, allowing us to begin the process of analyzing the data and answering the research questions in more detail instead of skipping over the data within this platform.

4 Results

In this chapter we are diving deeper into the RQ by analysing the targeted data and we will reflect on the implications of our findings. We further reflect on the implementation described in Section 3.2 and its capabilities as well as limitations. In general, all the visualizations presented in this chapter are part of the Jupyter Notebooks from the artio-insight repository [34] that is connected to the `UDM` schema of the corresponding database deployed on the University of Bern infrastructure. The aggregation and historization before the `UDM` is done by the go-db-etl [7] replication package that is used to collect and historize data from multiple sources in a single database.

The foundational data for the analysis within this thesis has been collected starting in April 2025 and includes data up until the end of April 2026, which means that we have been able to collect and analyze data from the Nostr ecosystem for a period of approximately 13 months.

4.1 Relay Landscape RQ1

RQ1
How diverse is the *relay landscape* based on ...
- RQ1.1 Unique operator pubkeys (`NIP11`)
- RQ1.2 Software stack and version (`NIP11`)
- RQ1.3 Supported NIPs and their combinations (`NIP11`)

In order to answer the first Research Question we analyzed the relay landscape by looking at the relays discovered through the artio-miner 3.2.2 and using `NIP65` events with the corresponding `NIPnip-11` documents collected from those relays (cf. Listing 2.3). The most important dimensions for this case can be found in the relay, relaynip and nip tables of the `UDM` schema in Figure 3.3. The analysis shows that not every relay that is advertised by users through their clients is actually reachable for everyone. One of the discovery runs executed we found a total of 8420 relays. Those can be categorized into the following categories by looking at their IP Address/Hostname in Table 4.1 and Figure 4.1.

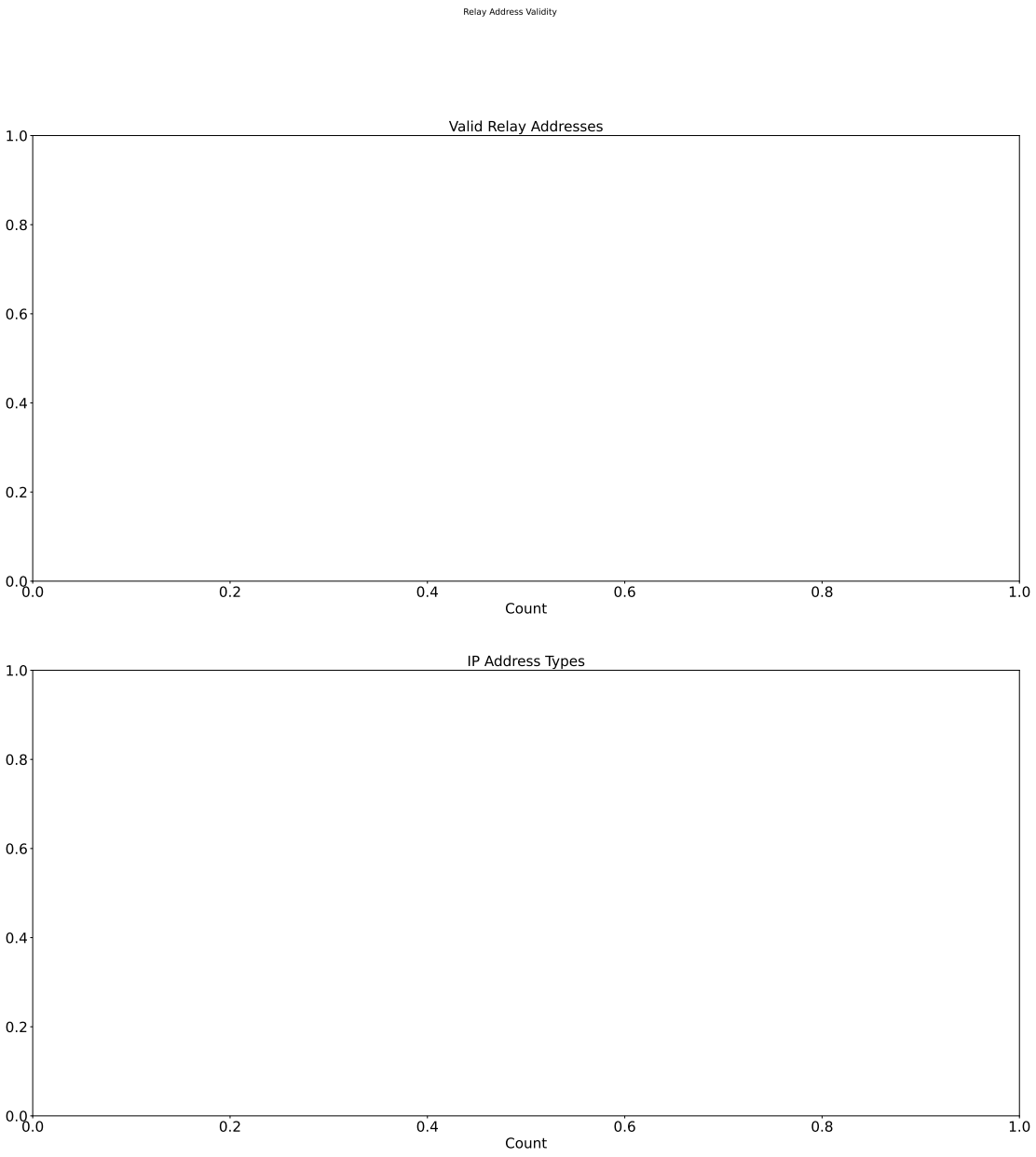


Figure 4.1: Relay Validity And Reasons For Invalidity

Validity Reason	Count
Valid Relays	3295
Unresolvable DNS Hostname	3832
Relays with TOR address [41]	621
Private IP Address [42]	231
Invalid URL [43]	194
Loopback IP address [44]	130
Carrier Grade NAT [45]	117

Table 4.1: Relay Validity Reasons

These are distinct numbers and do not represent multiple occurrences of the same relay. It shows that the majority of relays advertised by users ($\sim 63\%$) are not reachable from all vantage points, independent of DNS resolution issues specific to individual users (e.g., due to differing nameservers). [1]. We can only make a limited statement about DNS resolution failures since we only resolved the hostnames from a single vantage point within the network of the University of Bern. Within this network we do not control the DNS infrastructure and the firewall thus cannot guarantee that the results are not influenced by DNS resolution issues specific to this network.

Through the discovery of the relays using NIP65 we were able to collect NIP11 documents from the valid relays. Out of those 8420 relays we were not able to try to collect the NIP11 document (cf. Listing 2.3) from 1608 relays. This may be explained by several reasons. First of all, the NIP11 is not a mandatory NIP and relays might choose to not implement it. From a client perspective this makes interaction with relays much more difficult, since it is not officially known what feature a relay supports. Another reason might be that the relay is temporarily down or unreachable. NIP65 messages do not account for the reachability of a relay, only that it is advertised by a user through a client. NIP65 does not require a relay to be reachable for it to be included in a user's relay list. The last and most simple attempt in explaining this large number of missing NIP11 documents is that the relay simply does not implement the NIP11 endpoint correctly. Since NIP11 requires a specific HTTP Header to be set, i.e., "`Accept:application/nostr+json`", some relays or proxies in front of relays might not correctly handle this header and drop or block the given traffic.

We will use the information gathered from the NIP11 documents to answer RQ1 in the following sections. Important to note is that the information that we use to answer RQ1 is only based on the NIP11 documents that we were able to collect and thus only based on the relays that provide a NIP11 document, which is a subset of the valid relays.

4.1.1 Operator Public Keys Distribution

To reach an answer to RQ1.1, we analyze the distribution of relays across different operators. The relay operator is identified by the operator public key that is listed in the NIP11 document of the relay. A public key is a unique identifier that is derived from a private key and is used in cryptographic systems to identify entities without revealing their actual identity. In the event distribution it is used to identify the author of an event, while in the relay context it is used to identify the operator of a relay.

We expect to see a large amount of relays that are operated by a small group of operators which are heavily invested in the Nostr ecosystem and the protocol. Keeping in mind other results where we see bigger clustering such as the software distribution we will look at a bit later, this would not be a surprising result

[1] Nameservers: ns1.unibe.ch, ns2.unibe.ch

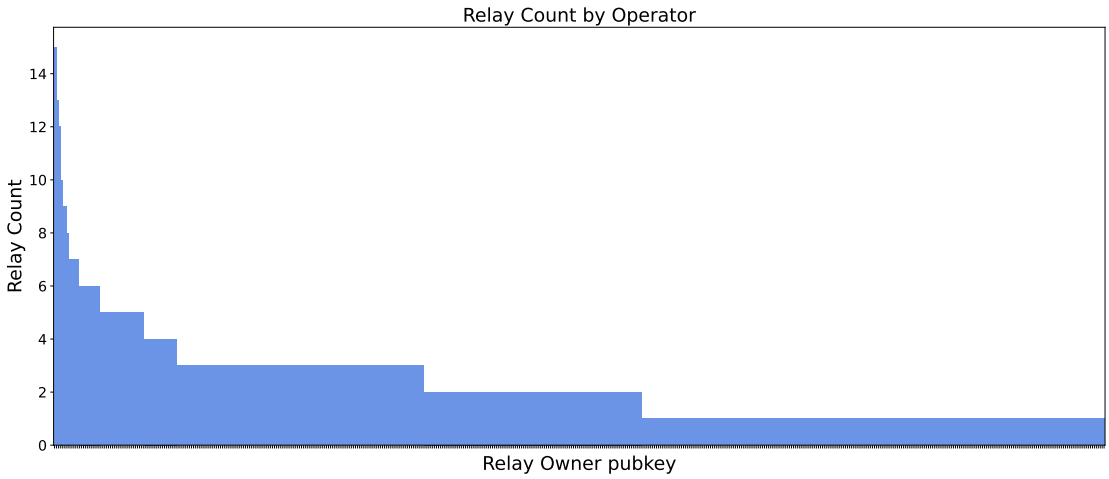


Figure 4.2: Amount Of Relays Grouped By Operator Pubkey

and would be in line with the general trend of software ecosystems being dominated by a small group of actors [2]. Figure 4.2 shows that there are indeed some operators that operate a large amount of relays, but the majority of operators only operate a single relay or two. While seeming to be a suffici8ent answer to RQ1.1 at first, but we need to take into account that any natural person can create multiple public keys that cannot be associated with each other. Additionally, the `NIP11` does not specify any way to verify the operator public key and thus it can be faked who owns a relay. As an example we could deploy a relay and take the public key of a well known operator (taken from the `NIP11` value of a relay) and just copy it into our own `NIP11` document, making it look as if the relay is operated by the well known operator, while in reality it is not. This could negatively influence the results and make it seem like the operator distribution is much more diverse than it actually is. Or manipulate it in a way that would be advantageous to a specific party. For the current time being we cannot make a statement about the actual impact of this factor and thus will leave it as is and take the results at face value. However, because identities can be freely created via new public keys, the observed operator distribution is not a reliable indicator of overall ecosystem diversity.

4.1.2 Software Distribution

To answer RQ1.2 we analyze the distribution of software used by the relays. In the initial look at the data we faced the problem that a not insignificant amount of relays did not provide a software name but rather a blank string. To achieve a more accurate result and answer to the software distribution we excluded any blank or null values from the analysis to avoid skewing the results. After excluding such values we arrive at the results visible in figure 4.3.

Compared to the diversity found in the operator distribution, the software distribution is much more concentrated where three major software stacks are used by a large part of relays. Namely, these are:

- **hoyetech/strfry** [46] written in C++, C and Perl. GPL 3.0 License. (38% of relays)
- **gheartsfield/nostr-rs-relay** [47] written in Rust. MIT License. (26% of relays)
- **bitvora/haven** [48] written in Go. MIT License. (6% of relays)

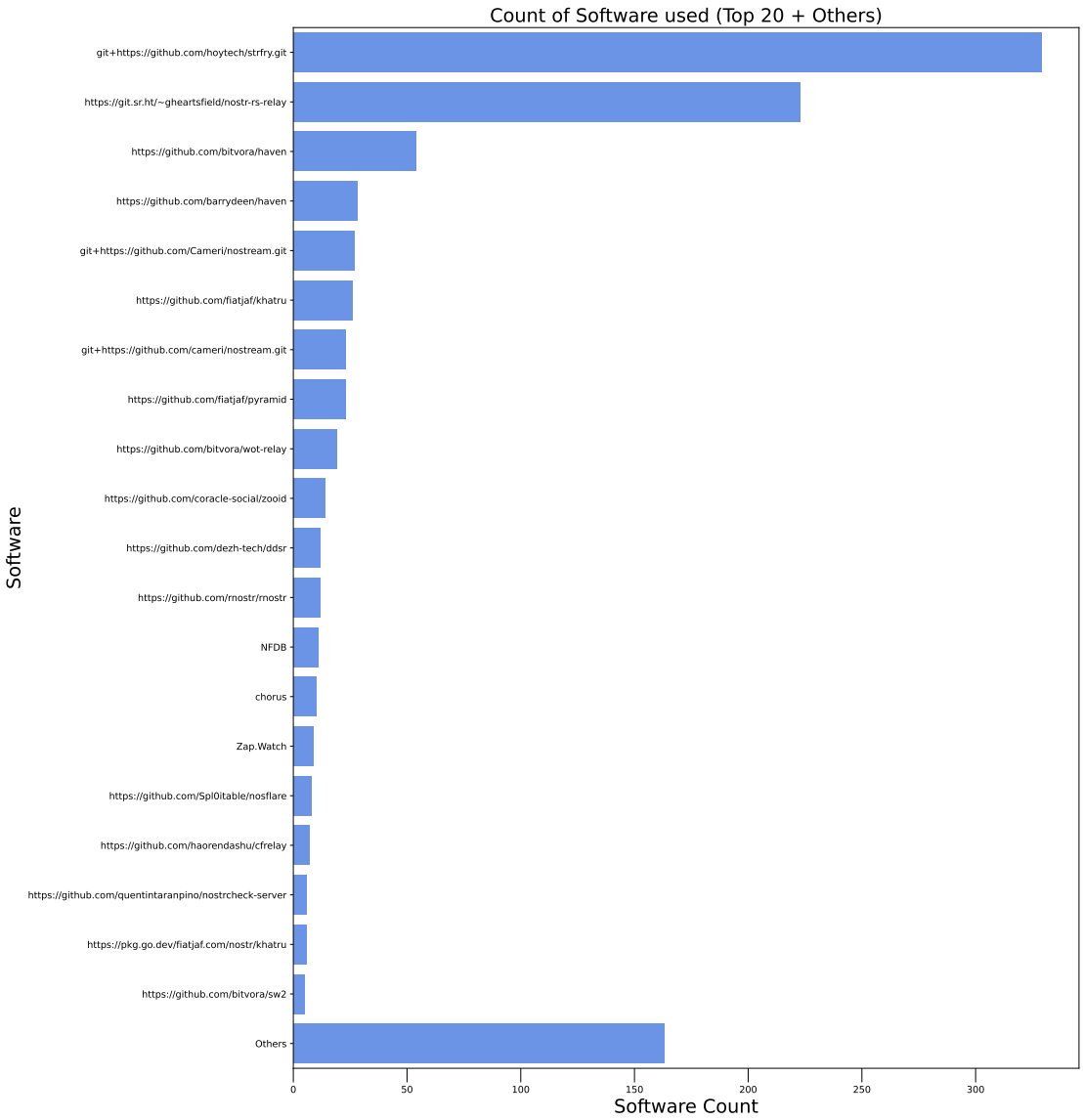


Figure 4.3: Amount Of Relays Grouped By Software (Top 20 And Others)

These three software stacks also represent some of the currently most popular programming languages. It thus shows that the Nostr ecosystem may not be diverse in the concrete software implementations used, but the implementations themselves are available in different programming languages. First proving that the programming language is not a requirement for the protocol to be functionally implemented and second showing that the community-driven paradigm [1, 2] is not leading to a monoculture of languages but offering variety leading to an easier entry point for developers with different programming language preferences.

All of the three software implementations are open source under either the GPL 3.0 or MIT License, allowing for possible clone-and-own [11, 49] variants and thus a more diverse software ecosystem. Due to the exclusion of missing software values, it remains unclear how many relays run custom, modified, or entirely different software than indicated in their NIP11 documents, as the reported information may not reflect the actual technology stack in use. In order to answer this question we would need to do a deeper analysis of the relays and their software stack, which is not possible without actually having access to the relay infrastructure and the binary files (or source code) running on it. For the time being we will take the distribution as is and leave the question of custom or altered software implementations for future work.

4.1.3 NIP Distribution

Now for answering RQ1.3 we analyze the distribution of supported NIPs and their combinations to get a better understanding of the diversity in terms of features and functionality that the relays are offering.

Out of the 1608 NIP11 documents collected from valid relays we are able to analyze the advertised supported NIPs. The supported NIPs used, are the ones shown in Listing 2.3

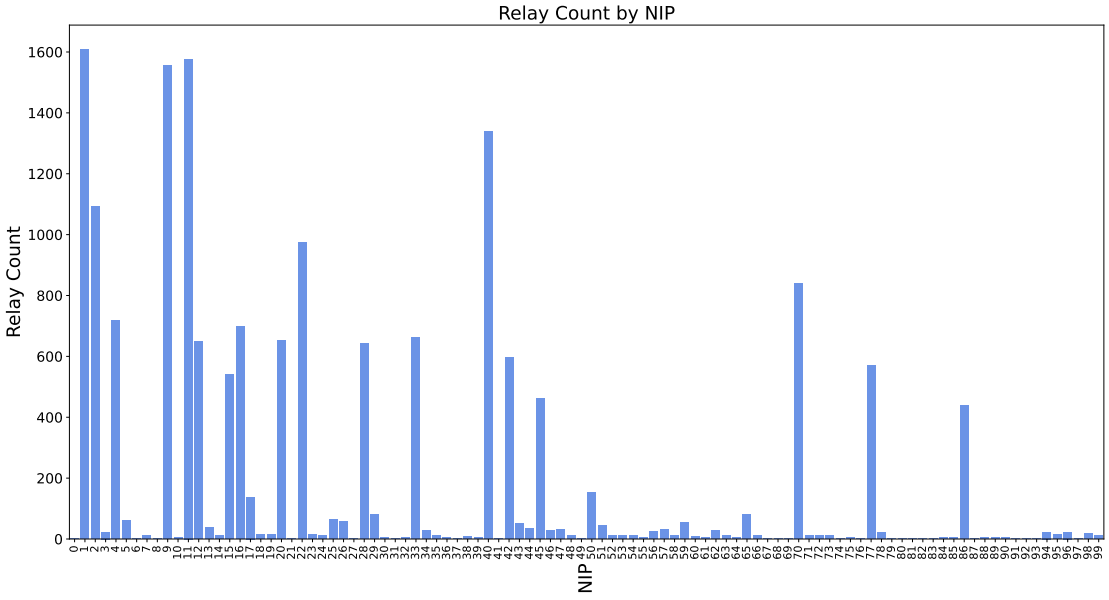


Figure 4.4: Amount Of Relays Grouped By Supported (Advertised) NIPs

The clear prominent NIPs are:

- NIP01 that is the core protocol and mandatory for all relays.

- NIP11 will be satisfied by all discovered relays since we only know about supported NIPs through the NIP11.
- NIP09 that defines the event deletion request, which is a feature most social-network-akin applications tend to support.
- NIP40 that defines an expiration timestamp for events, which might be useful to realize features such as stories (e.g. Instagram or WhatsApp stories).

Interestingly there are only a minor amount of relays that support NIP65. If we would assume that all relays and clients would strictly adhere to the NIP specification of the relays we should not be able to find any events of kind 10002 (the event kind defined in NIP65) on our relays that do not support NIP65. However, this was not the case: during development, our relays lacked NIP65 support, yet events of kind 10002 were still observed on them. A possible explanation for this 'unwanted traffic is the fact that NIP65 does not require any specific handling of the corresponding events on the relay side, but only on client side. The information that NIP65 is propagating is contained in the content field of the event, where the value is saved in the form of a JSON object as defined in NIP01.

For other cases it might be different, but it is a clear indication that the slim design of NIP-01 NIP01 enables clients to support NIPs without the actual need for implementation work on the relay side.

4.1.4 NIP Correlation

Given the diversity of supported NIPs and the fact that some NIPs are more popular than others we wanted to investigate the correlation between supported NIPs to answer Research Question RQ1.3. The result of the correlation calculation is visualized in Figure 4.5.

A first look at the matrix verifies assumptions that there are indeed groups of NIPs that are often supported together but also shows some contradicting results. We would have expected to see a high correlation value from nearly any NIP to NIP01, since it is the core protocol. However it seems taht a lot of relays do not bother to include NIP01 in their supported NIPs, even though they are actually supporting it since it is a mandatory NIP for all relays and the support for NIP11 implies that they are supporting NIP01. Other factors show that NIPs as NIP30 that specifies the handling of unknown event kinds and NIP89 are clearly linked. NIP89 describes two new event kinds that provide a way for client to discover applications that can handle unknown event kinds. This means, that when supporting NIP89, that NIP30 is actually needed since it is the technical groundwork and thus proves that there are some NIPs that are clearly linked in terms of support, implementation and functionality.

NIP Subset Correlation Since the large amount of NIPs that are present in the matrix, a reduction of the matrix to a subset of NIPs paints a clearer picture of the correlation between NIPs, as shown in Listing 4.1. The subset of NIPs is chosen based on the amount of relays supporting them, where we included the top 20 most supported NIPs and NIP65.

```
1 [1, 2, 4, 9, 10, 11, 12, 15, 17, 20, 22, 28, 33, 40, 42, 45, 50, 65, 70, 77, 86]
```

Listing 4.1: Top 20 Most Supported NIPs And NIP-65

NIP65 is not in the top 20 most supported NIPs, but we included it since it is the NIP that we used for the discovery of relays and thus is an important part of the ecosystem. Figure 4.6 shows a more nuanced picture of the correlation between NIPs, than the full matrix. We can now identify a trend, that the subset of NIPs under NIP40 are often supported together, with the exception of NIP10, NIP17, NIP50 and NIP65.

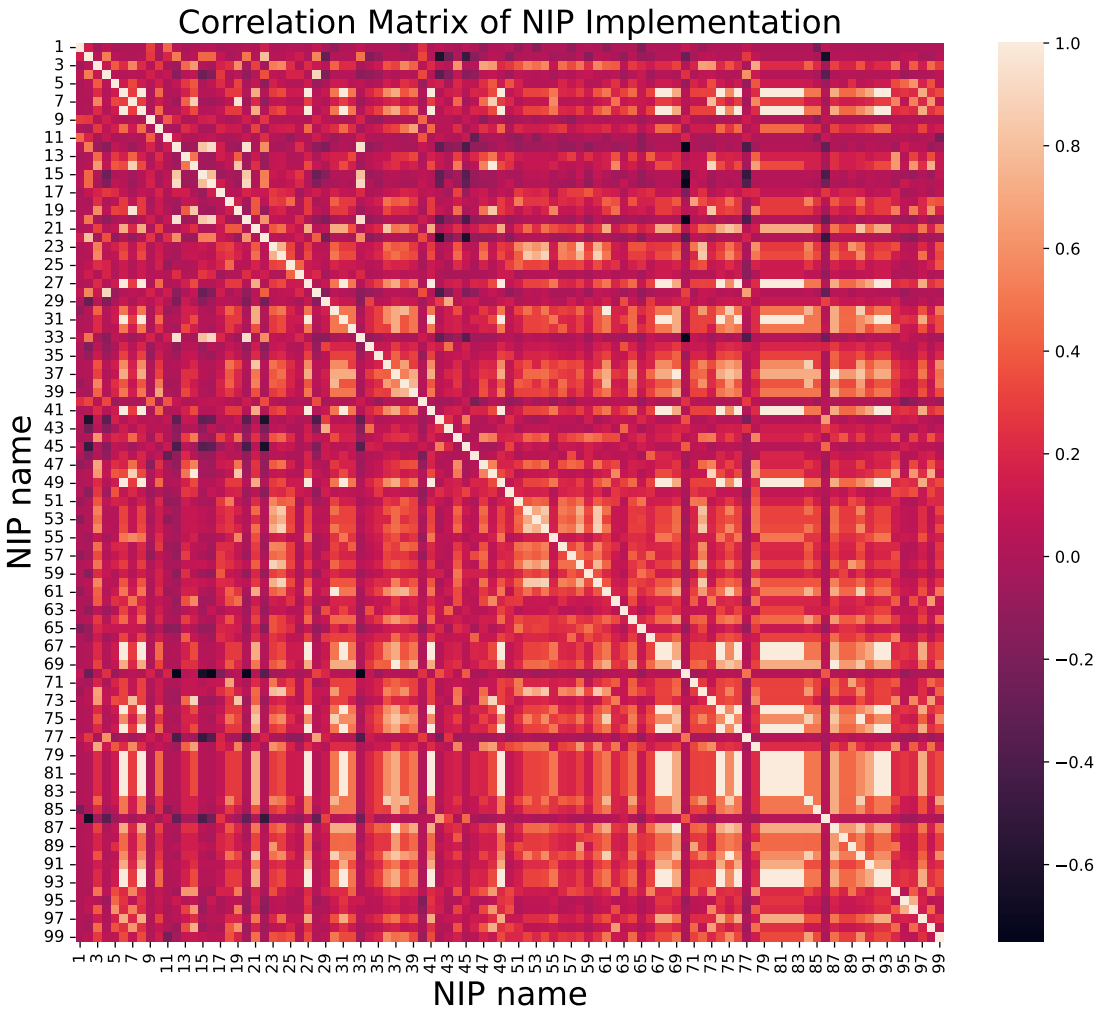


Figure 4.5: Nostr NIP Correlation Matrix

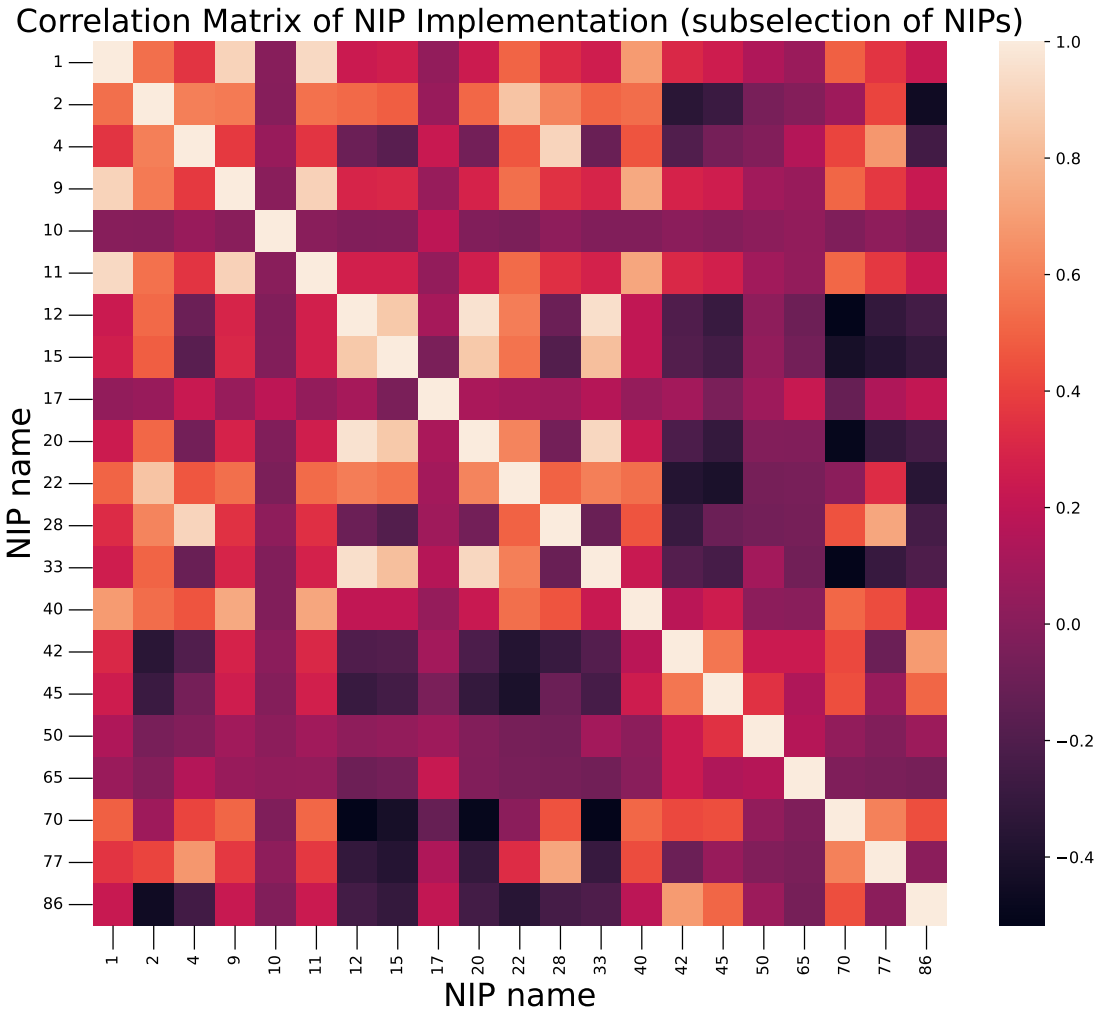


Figure 4.6: Nostr NIP Correlation Matrix With Reduced Set Of NIPs

The highest correlation value can also be optained by converting the correlation matrix into a series. Transforming the correlation matrix into a series representation yields the maximum correlation. After removing self-correlations and duplicate pairs (e.g., NIP01→NIP09 and NIP09→NIP01), we identify the pairs with maximal correlation.

Table 4.2 and Table 4.3 show the highest and lowest correlation values, respectively. Initially, we found an outlier in the form of NIP10 that had a correlation value of 1.0 (the highest possible value) with nearly all other NIPs. The reason for that case was that NIP10 has only been found in one relay and thus the correlation value is not meaningful. For the top and bottom correlation values we have excluded the specific relay from the list. We can still see, however, that some NIPs have a correlation value of 1.0 with other NIPs, which means that they are always supported together. These patterns might deviate when some NIPs are only supported by a single or minor amount of relays since the correlation value is not meaningful in those cases. In a broader view on the tables they show that there indeed is a spectrum of software variations present and we show it is not only a theoretical possibility of software variation but rather a real and active spectrium within the Nostr ecosystem.

4.1.5 Summary

This first section has given us a good overview and lets us answer the first research question about the relay landscape we can summarize the findings as follows. The relay landscape is much more complex than we initially expected. While the amount of information about relays is high, so is the number of relays that are not actually reachable for our analysis and thus cannot be considered to be an active part of the ecosystem.

Unique Operator Pubkeys RQ1.1 The amount of unique operators and their distribution is diverse and the amount of relays per owner is smaller than expected. It is a clear indication that the relay ecosystem is not dominated by a small group of operators, but rather a large amount of operators with a small amount of relays each.

Software Stack And Version RQ1.2 The software stack and version distribution is much more concentrated than the operator distribution. The amount of around 90 software is, however, still high considering that we also have some relays that do not provide any information about the software they are running or are not providing any NIP11 document at all. The top three software stacks indicate that the protocol does not have any favoritism towards a specific programming language and that the community-driven approach is not leading to a monoculture of languages but offering variety leading to an easier entry point for developers with different programming language preferences.

Supported NIPs And Their Combination RQ1.3 The supported NIPs it self show that there are some clear favorites in the ecosystem, with (expectedly) NIP01 being the most supported NIP, since it is the core protocol. This result is not surprising and is actually a good sanity check for our data collection and analysis. The correlation between supported NIPs shows that there are some NIPs that are often supported together, but also some that are not or even contradict each other. Some of these trends can be explained by the fact that some NIPs build on top of each other, while others might be just coincidental or based on the preferences of the users.

Overall the results display a rather high diversity in the relay landscape with a large amount of operators and a large amount of different software stacks and versions, but also some clear favorites in the ecosystem. The diversity is growing and becoming more complex at a high rate in terms of the amount of relays, operators and nip operators. However, over the time of this research, the rate of growth could not be measured for the software but we did not see any significant change in the software distribution.

NIP Name	Second NIP Name	Correlation
6	8	1.000 000
6	27	1.000 000
6	31	1.000 000
6	41	1.000 000
6	49	1.000 000
6	67	1.000 000
6	68	1.000 000
6	74	1.000 000
6	76	1.000 000
6	79	1.000 000
6	80	1.000 000
6	81	1.000 000
6	82	1.000 000
6	83	1.000 000
6	92	1.000 000
6	93	1.000 000
8	27	1.000 000
8	31	1.000 000
8	41	1.000 000
8	49	1.000 000

Table 4.2: Top Correlating NIPs

NIP Name	Second NIP Name	0
12	70	-0.750 574
33	70	-0.749 611
20	70	-0.736 304
16	70	-0.730 546
2	86	-0.649 751
15	70	-0.641 744
22	45	-0.635 065
22	42	-0.624 445
2	42	-0.618 332
22	86	-0.508 523
15	77	-0.493 770
2	45	-0.479 560
20	45	-0.414 119
28	42	-0.407 137
12	77	-0.401 367
15	86	-0.401 095
20	77	-0.395 927
12	45	-0.395 352
33	77	-0.392 658
16	77	-0.362 680

Table 4.3: Bottom Correlating NIPs

4.2 Event Landscape *RQ2*

Research Question 2 RQ2
How diverse is the *event landscape* based on ...
- RQ2.1 Message types (NIP01)
- RQ2.2 Message kinds (cf. nostr.dev)

Given the already discovered anomalies in the Nostr relay ecosystem, we further analyzed the occurrence of events on our own relays. The main data dimensions is the event table which we will expand using the `nipkind` and `NIP` table. Using the event kind to nip mapping found in [24] we can have a look at the distribution of messages by their corresponding NIP. This will allow us to approach RQ2 and RQ3 by looking at the distribution of events corresponding to supported and unsupported NIPs on our relays. This section focuses on the results associated to RQ2.

4.2.1 Message Types

Now for the analysis of the RQ RQ2.1 we can not rely on the event tables of our relays since some message types are not directly stored as events on our relays. Looking back at the NIP01 we can see that there are the following message types defined:

- From Client to Relay
  - `EVENT` which is used to publish events
  - `REQ` to request events with certain filters
  - `CLOSE` to close any subscriptions (filters, where implemented)
- From Relay to Client
  - `EVENT` the requested events
  - `OK` indication that a message has been accepted or rejected by the relay
  - `NOTICE` to send human readable messages to the client
  - `EOSE` used to indicate the end of transmission for stored events.
  - `CLOSED` indication that a subscription has been closed server side.

An additional message type is the **COUNT** message type that is introduced in NIP45 and is used to query the relays for the amount of events that match a given filter. A key challenge in our analysis of message types is that, apart from `EVENT` messages, no other types are persisted in the relay database by default. In standard relay implementations, this renders analysis of the full message type spectrum infeasible. To address this, our *artio-relay* implementation [23] includes a logging mechanism that preserves all relevant events within the relay. This also includes the `REQ` message type that should be answered from the relay with the event message type (if there are any found), followed by the EOSE message.

The actual amount of opened connections is not being logged in the current implementation. However, Figure 4.7 indicates that the biggest amount of messages handled by our relays are the `REQ` message type. This is expected as clients will be requesting events from the relay and the relay will be responding with the `EVENT` message type. The second biggest amount of messages is the `CLOSE` message type, which is also expected as a number 2 since every connection needs to be closed eventually. On the third place there is the `EVENT` message type where we can clearly see that the amount of events handled is much smaller than

the amount of REQ and CLOSE message types, meaning our relay was handling more outgoing message than incoming messages.

However, an interesting finding is the fact of message types that we did initially not know about up until we found them in our logs. ur analysis revealed previously undocumented message types in the logs. In particular, we observed NEG-CLOSE and NEG-OPEN, which are introduced in NIP77 and are used for syncing events. We did only received a very small amount of messages of this kind (just over 700) and the amount of NEG-OPEN messages is nearly identical. The comparison of the amount these two message types over both our relays as visible in Table 4.4 shows that the amount of messages compared to each other is similar to the ratio between other message types comparing the two relays.

index	Relay	Content	Count
0	relay.artio.inf.unibe.ch	AUTH	49 815
1	relay.artio.inf.unibe.ch	CLOSE	3 605 376
2	relay.artio.inf.unibe.ch	COUNT	101
3	relay.artio.inf.unibe.ch	EVENT	1 381 514
4	relay.artio.inf.unibe.ch	NEG-CLOSE	626
5	relay.artio.inf.unibe.ch	NEG-OPEN	654
6	relay.artio.inf.unibe.ch	REQ	5 265 042
7	relay.artiostr.ch	AUTH	240 972
8	relay.artiostr.ch	CLOSE	6 884 529
9	relay.artiostr.ch	COUNT	695
10	relay.artiostr.ch	EVENT	4 985 647
11	relay.artiostr.ch	NEG-CLOSE	905
12	relay.artiostr.ch	NEG-OPEN	966
13	relay.artiostr.ch	REQ	16 391 983

Table 4.4: Message Types handled by our Relays

This suggests that a single user or a small set of actors is attempting to synchronize events, despite the absence of this functionality in our artio-relay. Its occurrence on both our relay instances is likely explained by their shared software and operator public key. A deeper investigation into other relays is not possible since we do not have access to their logs, assumed the used software stacks do even have the correct logs, to see if such unsupported message types are also being sent to other relays.

Another way to track the usage of our relays is the monitoring via prometheus [25] and grafana [35] that we have deployed for our relays. In order to see more the amount of incoming and outgoing messages we implemented a customer prometheus exporter that tracks the amount of incoming and outgoing messages by type and exposes this data publicly. For answering our RQs we are not actively using this data, but it is interesting to see the amount of messages handled by our relays in real time and to see the distribution of message types in real time. For better visualization we integrated it into our grafana instance, created a dashboard for it and made it publicly available [26].

4.2.2 Message kinds

Figure 4.8 shows that a great amount of events stored on our relays do not directly correspond to a NIP, but rather to ranges of event kinds that are reserved for various usages such as (parametrized) replaceable events or ephemeral events. Figure 4.8 suggests a first answer to the RQ2.2 in a way that the nip usage is

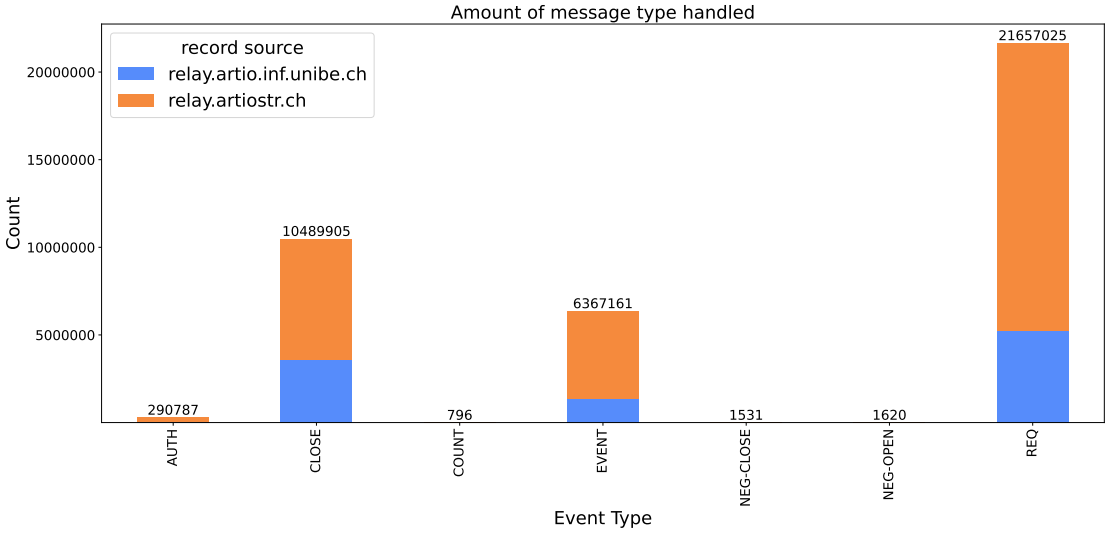


Figure 4.7: Message Type Handling By Type And Split By Relay

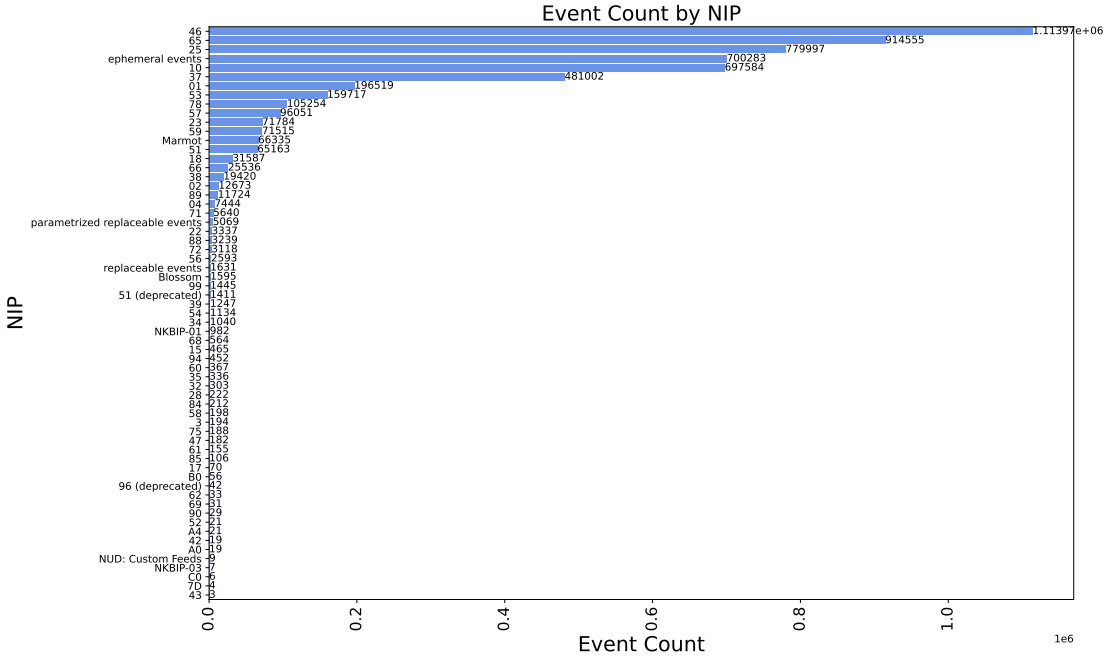


Figure 4.8: Event Count By Corresponding NIP

diverse. There are some clear winners in the distribution such as the NIP65, NIP25, and NIP46 events. Several NIPs dominate the distribution, notably NIP65, NIP25, and NIP46. NIP65 and NIP25 each define a single event kind (10002 and 7, respectively), which facilitates their identification in the event distribution. The usage of NIP25 is something that we will revisit again in Section 4.3 since it is not supported on our relays but still used. The ranges (e.g. ephemeral events) are defined in NIP01 and are not directly tied to a specific NIP.

Start Kind Range	End Kind Range	Description
1000	10000	regular events expected to be stored by relays
10000	20000	replaceable events
20000	30000	ephemeral events
30000	40000	adressable events

Table 4.5: Kind Ranges Defined In NIP01

However, some kinds are tied to a NIP and are also included in a range. The perfect example is NIP65 which makes use of the kind 10002, which is part of the regular events range (cf.Section 4.2.2) and still is tied to a specific NIP and is visible as such in our visualization. The amount of ephemeral events is interesting and will revisited in Section 4.3 about NIP conformity analysis. This case is already handled in the sql query used for fetching the data for visualization where we map the event kind to the NIP if we can find one, otherwise we check if it is part of a range we could map it to. Absolute counts can be misleading, as the number of events grows over time and may obscure usage trends. We therefore analyze the events handled directly by our relays and their occurrence over time.

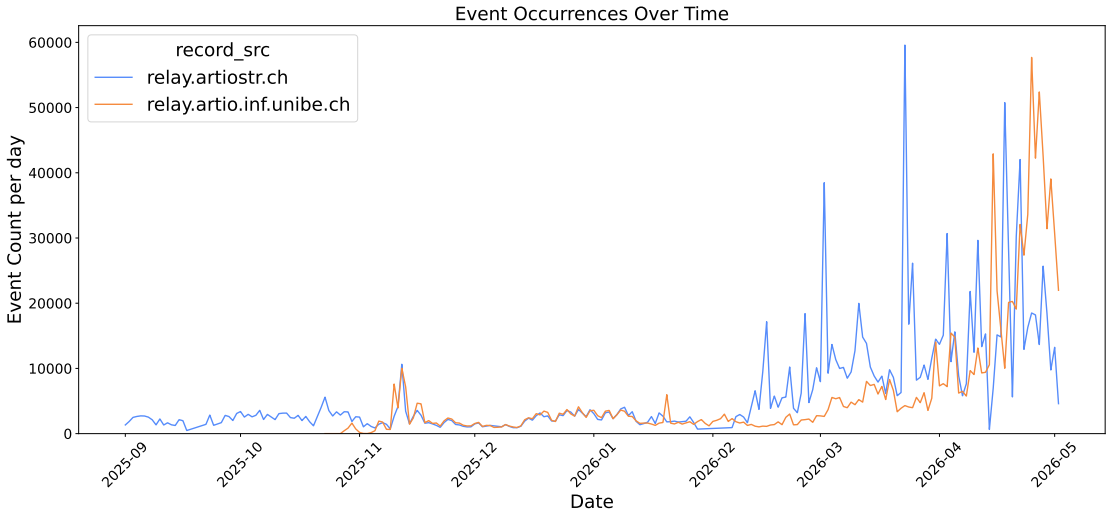


Figure 4.9: Event Occurrence Over Time By Relay Per Day

As a first insight and sanity check we can see in Figure 4.9 that the amount of events stored per day on our relays is not constant over time but fluctuates notably. In our relay databases we are not deleting any events, but only marking them as deleted and handling the case of event deletion requested from the clients in the artio-relay implementation [23]. For the end user of our relays this means that a deleted event is not visible anymore. As for all RQs, Figure 4.9 also includes deleted events, since they are still interesting for usage analysis and if they would be excluded the amount of events over time might be misleading and lead towards

wrong conclusions. It is clearly visible that the `relay.artiostr.ch` (referred as `Homelab Relay`) has a much higher amount of events stored compared to the `relay.artio.inf.unibe.ch` (referred as `UNIBE Relay`) and the production relay is indeed not running for as long as the artiostr.ch relay. Next we look at the distribution of events corresponding to nips over time.

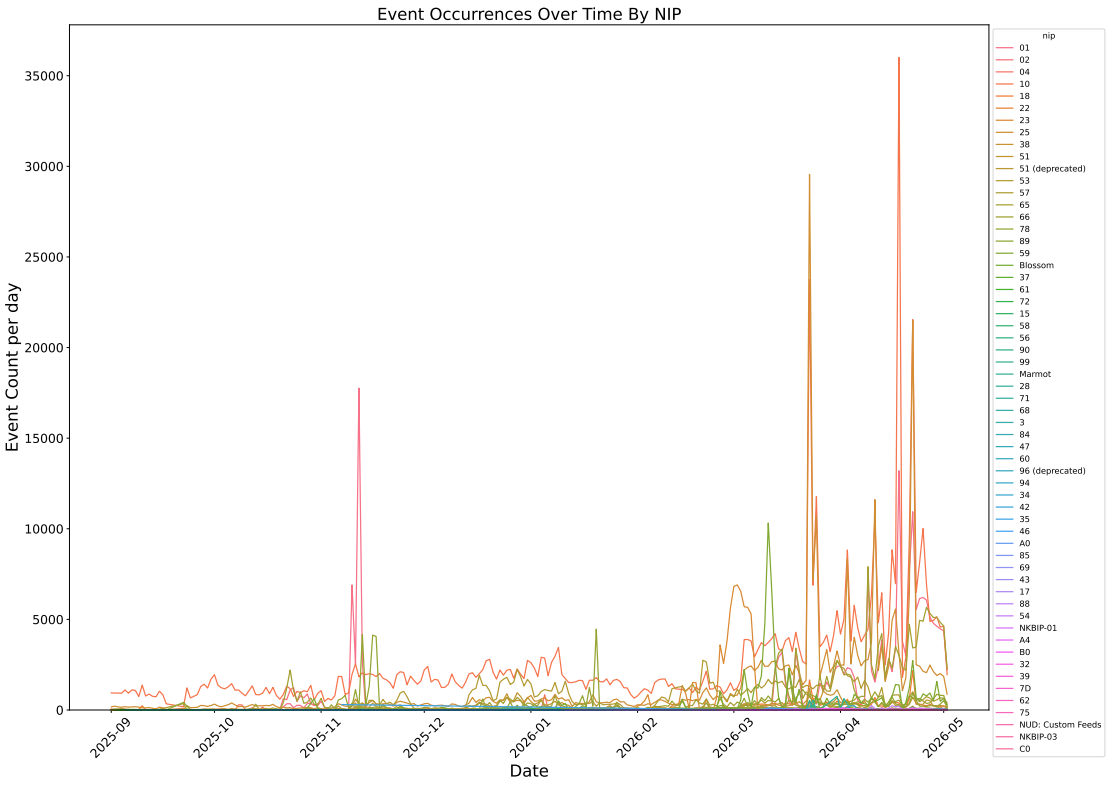


Figure 4.10: Event Occurrence Over Time By NIP Per Day

As indicated by our NIP-11 document in Listing 2.3, a preliminary observation relevant to RQ3 emerges: The temporal distribution exhibits short-lived spikes (notably in March 2026), which also appear in NIP-specific event distributions. These spikes reflect temporary increases in specific event types rather than sustained deviations. Given their short duration, they are best interpreted as normal fluctuations, potentially caused by automated activity or short-term probing of our relay by others. We are not aware of any user specifically testing our own relays and will thus look at this data as an normal fluctuation in the distribution.

The distribution of events corresponding to NIPs over time does not show a clear trend and is quite fluctuating, which makes it hard to draw conclusions from it apart from the absence of trends. However, the fact that no trend can be spotted still gives us insight into the fact that there is no pattern visible in the distribution of events over time. During the observed time span in Figure 4.9, there have been multiple modifications and enhancements to the artio-relay implementation. An example of such change is the addition of support for `NIP42` that handles the authentication, which has been introduced in the pull request [50]. Not publicly documented is the deployment of the given software version to our two relays on the same day. Revisiting Figure 4.10, no change in the occurrence of events corresponding to NIPs can

be observed within several days since the corresponding changes were applied.

4.2.3 Summary

For the answer of RQ2 we can say that there is a diverse distribution of events corresponding to NIPs on our relays, but no clear trend can be observed in the distribution of events over time. However, this analysis lays the groundwork for RQ3, particularly NIP conformity and the distribution of supported versus unsupported NIPs on our relays.

Message Types RQ2.1 The amount of message types handled by our relays is roughly as diverse as expected, with the `REQ` message type being the most handled message type, followed by the `CLOSE` and `EVENT` message types. However, we did see some unsupported message types being handled by our relays (NEG-CLOSE and NEG-OPEN), which is an interesting finding that we will carry over into the next section about NIP conformity analysis Section 4.3. Overall the properly handled types do not show any anomalies and are highly expected given their introduction in the `NIP01` and the expected usage of a relay.

Message Kinds RQ2.2 Compared to the message types, the distribution of events kinds is much more diverse than expected and does not show any clear trend over time. What we can say, is that there are some clear favourites in the kinds, that are tied to specific NIPs such as the `NIP65` and `NIP25`, but there is also a huge amount of events that does not correspond to a specific NIP or to NIPs that are not supported on our relays. This is a clear indication that the RQ3 about NIP conformity is going to be an interesting one to analyze, and violations are clearly visible in the distribution of event kinds. Overall the event landscape is highly diverse in terms of message kinds, but quite strict in terms of message types.

4.3 NIP Conformity *RQ3*

After the initial look into event distribution we can observe that apart from the events based on the ranges defined in `NIP01` and the NIPs, that we officially support on our relays (according to Listing 2.3) a great amount of other events is being stored on our relays.

Research Question 3 RQ3
How NIP-conform are messages propagated in the network?
- RQ3.1 How many officially non-supported-kind messages did we receive (e.g. we receive 10002 despite not supporting NIP65)?
- RQ3.2 To what extent do relays fail to handle message kinds of allegedly supported NIPs?

4.3.1 NIP Conformity

These events correspond to NIPs that we do not officially support on our relays. As shown in Section 4.2, many events on our relays do correspond to NIPs that are not officially supported by us, meaning that the information provided in `NIP11` is often ignored by clients. A first look at the supported vs. unsupported NIP event occurrence in Figure 4.11 shows that the majority of events on our relays correspond to unsupported NIPs.

But as mentioned before, large parts of the events corresponds to ranges of kinds that are not directly tied to a specific NIP, but to a pre-reserved range of variable uses, for which we cannot definitively say if they are supported or unsupported. Excluding these events from the mentioned NIP ranges, we arrive at the distribution shown in Figure 4.13. Comparing the two numbers between the unsupported with and without

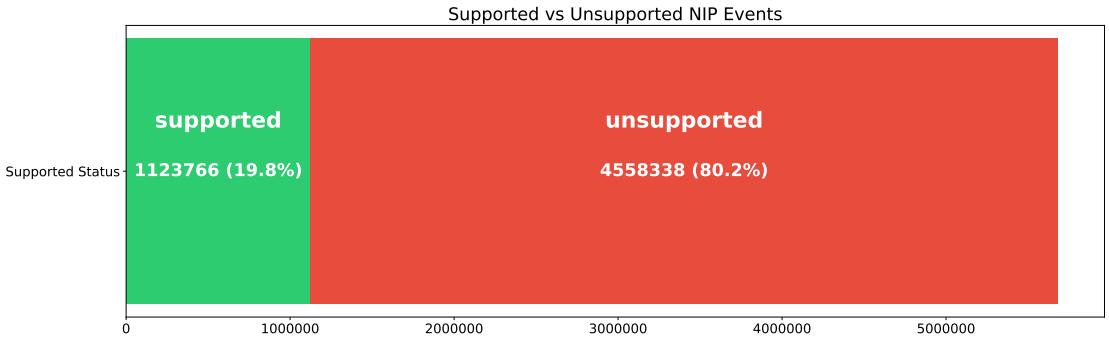


Figure 4.11: Supported Vs Unsupported NIP Event Occurrence

ranges we only see a decrease of the unsupported NIPs from 80% to 77%, which means that the majority of events on our relays still correspond to unsupported NIPs for which we can map them directly to a specific NIP and thus are clearly identifiable as unsupported kinds.

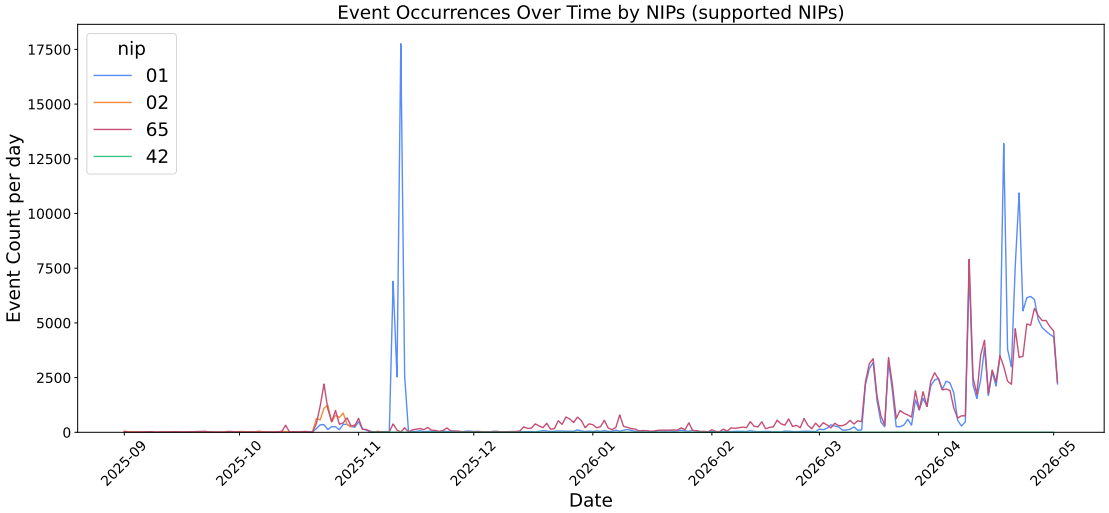


Figure 4.12: Supported Event Occurrence Over Time By NIPs

This result can answer RQ3 in the sense that there is a non-negligible amount of events stored on our relays that correspond to NIPs that we do not officially support. An explanation for this finding can be that clients do not check for the supported NIPs of a relay before sending events to it. As mentioned earlier in Section 4.1 the `NIP11` is not mandatory and takes some of the blame off of the clients, since for a relay that does not implement `NIP11`, there is no way for the client to know what is supported and what is not. For our own relays, where we are certain, that `NIP11` is properly implemented, there is no excuse for clients to not check for supported NIPs before sending events to our relays, from our point of view.

Overall we can find that 80% of the events on our relays correspond to unsupported NIPs, which is a clear indication that there is a large amount of non-conformity to the supported NIPs on our relays. This is a total of 4 556 663 unsupported events, compared to 1 123 752 supported events.

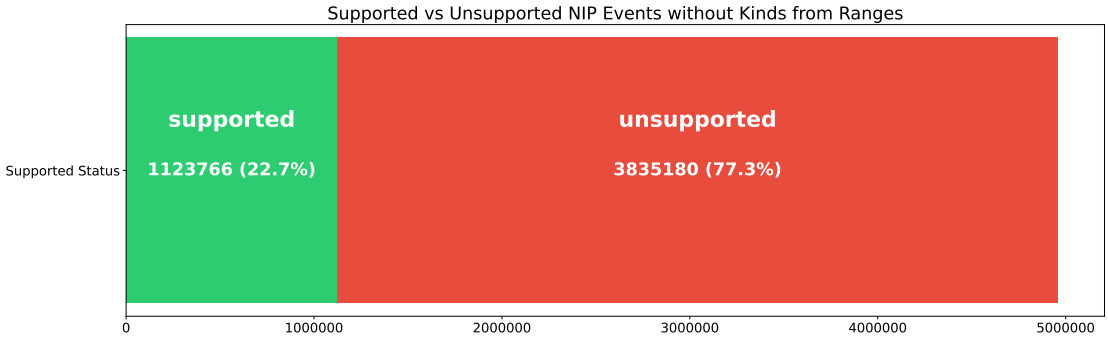


Figure 4.13: Supported Vs Unsupported NIP Event Occurrence (Excluding NIP Ranges)

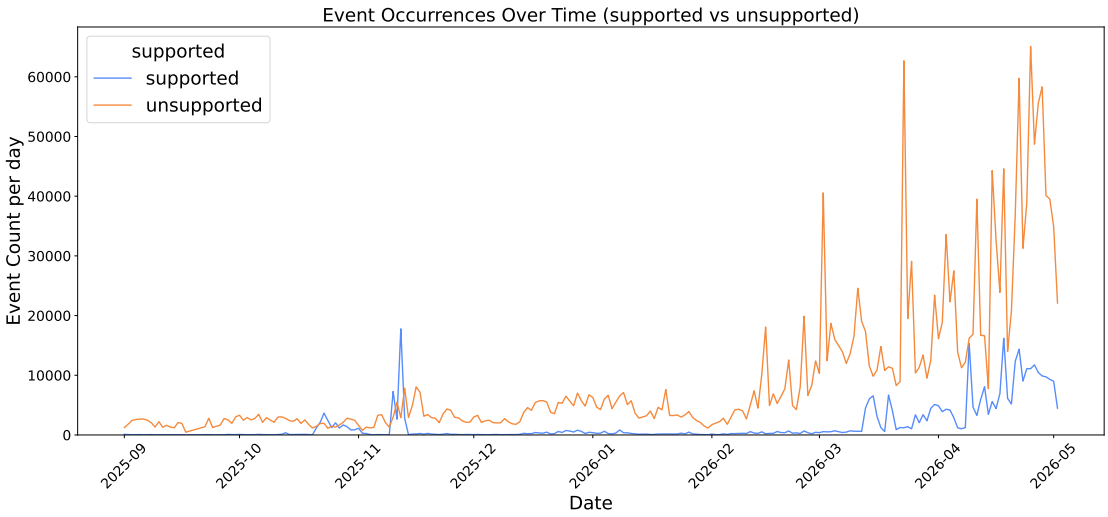


Figure 4.14: Event Occurrence Over Time Supported Vs Unsupported

4.3.2 NIP Handling Failure

The question of RQ3.2 is still open and requires some deeper explanation. For all events that correspond to unsupported NIPs, we can say that the publishing part has been handled correctly by our relays, since otherwise the events would not be stored on our relays at all. But handling does not only include the publishing part, but also the reading part of Nostr. For the reading part we need to make a distinction between the events that should be publicly available on the relay for everyone to read, events that should be replaced and others that should only be temporarily available or to a select group of users.

One example for the events that should only be available to a select group of users is when a relay requires authentication for reading events of a specific NIP and its kind. We have already seen in Section 4.2 that NIP40 is one of the more popular NIPs. To publish an event of any kind with an expiration timestamp in the tags will always be possible in our implementation, since we do not want to limit functionality by locking down on only known tags. This would violate the principles defined in the root of the NIP repository about rules being created when necessary and not creating rules that limit the protocol without a good reason [13]. But given our artio-relay instances do not advertise support for NIP40, we do not handle the expiration timestamp properly and thus the events that correspond to NIP40 are not actually expiring on our relays, but rather are stored as regular events.

Looking back at NIP40 we do not find it in the events that are present on our relays. This means that that never has an event of NIP40 been published to our relays, which is a sign of NIP conformity within clients. NIP40 requires some implementation to be done on the relay side, meaning that if relays do not advertise support for NIP40, clients are certain that the expiration will not be handled properly and thus might not chose to publish events of said kind to those relays. Another example that proves the point of NIP conformity is the case of NIP04 (encrypted direct messages) where the client must rely on proper relay side implementation.

Revisiting Section 4.2, several open questions arise regarding the validity of events that do not correspond to supported NIPs and related aspects. As illustrated by the example of NIP40, we observe no corresponding events on our relays, as unsupported NIPs cannot be handled without the necessary implementation details. In contrast, the third most frequent NIP by event count in Figure 4.8 is NIP25, which defines user reactions to events. One might now ask the reasonable question about the difference between NIP40 and NIP25, since they are both not supported on our relays but on is handled and the another one is not. The key difference is that NIP25 does not require any specific handling on the relay side, since the logic of the NIP is fully embedded in the content field of the event. The content field is a field which is actually not required to have any handling or reading done on by the relay. Any alteration of said field would inevitably lead to an invalid event signature, resulting in the event being rejected by all involved parties, that validate the events.

Likewise the reaction to an event uses the `event id` and `pubkey` to reference the main event including a hint to the relay where the main could be stored. As long as the main event exists, there might be some requests to our relay to fetch the reactions to said event. If the main event gets deleted but not the reaction to it as it might have been published to a different relay, the reaction will still be stored on our relays but will not be fetched by clients since the main event is no longer available. For the unlikely case that a user is first querying for reactions without having already queried the event, we will still return the reaction, but the client will not have any way to display it properly in relation to the main event due to its inexistence.

The same logical principle applies to the most popular NIP, i.e., NIP46, where the relay is solely used as an intermediate storage for exchanging cryptographic events. As long as the basic NIP01 is handled properly, the specific event kind is not blocked and the content field not altered, the events of NIP46 can be published and stored on our relays without any problem, even if we do not support NIP46 and thus do not have any specific handling for it. This explanation answers the question, because any error that might

occur due to the handling of the event on the relay side (returning reaction to non-existent event), can just be handled on the client side without any other implications.

Another facet to mention from the analysis conducted to answer RQ2 is the amount of ephemeral events stored on our relays. As stated in NIP01 ephemeral events are not expected to be stored by relays. It raises the question if we have some violation of NIP01 in our own implementation. While the investigation of this issue we found that some of the ephemeral events that are stored on our relays are actually associated with other NIPs that are not supported by our relays, such as NIP46 (Nostr Remote Signing [51]). While this means that the events are indeed not handled properly on our relays, for at least a part of the ephemeral events, we do not take the full blame since (once more) there is an explanation in users and clients not respecting our NIP11 document and sending events of unsupported NIPs to our relays. However, it raises the immediate question on how to approach and interpret this finding. On the one hand it is a clear violation of NIP01 to store these ephemeral events, when we do not have knowledge about the (maybe corresponding) NIPs and thus do not know how to handle them properly. On the other hand, if we would drop or block the event we would violate the spirit of the acceptance criteria for NIPs where it is stated that it should be optional and backwards compatible and not make other relays stop working with the ones that choose to.

We will discuss the implications of these findings in a greater scope in Chapter 5, but for the time being we will just take it as a finding that as another proof that violations of NIPs are not only present in rare cases but rather can be observed on a larger, systematic scale and thus can be perceived as major violations of advertised and supported NIPs thus giving a direct answer to RQ3.1. Within the scope of RQ3, the protocol's design and backward-compatibility requirement imply that unsupported NIPs are not mishandled. Instead, events that require special handling (e.g., NIP40) are typically not observed.

4.3.3 Summary

Over all we find that from our own relays the majority of events we handle are not supported by our relays, meaning that they correspond to NIPs that we do not officially support. Overall this states a clear violation of the NIP11 and thus a non-conformity to the supported NIPs on our relays.

Non-Supported-Kind Messages RQ3.1 Overall we received over 4 million events that we are not officially supporting but do handle and store conforming to our relay specifications, which is a clear indication of non-conformity to the supported NIPs on our relays. From the perspective of proper handling, no definitive answer can be given. If a NIP is not supported, there is no obligation to handle it, as such support is not advertised and clients should not send corresponding events in the first place.

Failure To Handle Message Kinds RQ3.2 For the cases of NIPs that require special handling in the relay implementation, we do not fail to handle them as we do not receive such events from the clients. Our supposed explanation is that the clients are aware of NIPs that require relay-side handling and thus do not send events of such NIPs to our relays. For relays that we do not support, but receive anyways events for, we consider this handling as not failed, but rather handled properly since the clients decided to send events even though non-supporting it. Overall we do not fail to handle any events of supported NIPs, but cannot make a statement about the handling of unsupported NIPs, since we did not any events of unsupported NIPs that require specific handling on the relay side.

4.4 Usage Topology (RQ4)

Research Question 4 RQ4
How do usage trends manifest in the Nostr ecosystem?
- RQ4.1 Relay discovery by software
- RQ4.2 User to relay Usage patterns

For RQ4, we analyze the usage topology of our relays based on results from previous sections and additional analyses. The primary data source is the NIP65 analysis, which enables identification of trends in relay usage and software stacks. The key dimensions that we consider for the analysis of the usage topology are the relays, the users of those (NIP65) and the software stacks of those relays (NIP11). While the first three research questions focused on the characteristics of the ecosystem in a static way from the viewpoint of our own relays, the fourth research question focuses on the dynamic interactions between the different dimensions and how they influence each other. The key expansion in data selection therefore is the user dimension, which is not directly visible in the previous analysis but can be derived from the NIP65 messages by grouping over other dimensions.

From the NIP65 analysis we can first of all see that there are some relays that are advertised as being used by a large amount of users, while some relays are only advertised by fewer. Events from relays that require payment are excluded from this analysis.

4.4.1 Relay and Software Usage

Leveraging the data from the NIP65 analysis, we want to dive into the usage topology and focus on new insights that we can fetch from the relations generated by the `artio-miner` (cf. Section 3.2.2, [30]). First of all, we want to focus on just the relays and their software stacks.

Table 4.6 shows the top 20 relays by user popularity, measured by the number of users advertising each relay. The result of users per software in Table 4.7 is achieved by grouping the relays by their announced software and counting the distinct public keys that are advertising those relays. Using this distinct count for the users is important to avoid counting one user several times if they advertise their usage of given relays on multiple relays as this advertisement on multiple relays is expected in the main design of Nostr.

Compared to the Figure 4.3 the top software stacks by users are not identical to the top software stacks by relays, but there is a clear overlap. The top three software stacks by relays are only in place 2, 4 and 14 by users, indicating a clear difference. While it is straightforward to deploy several relays using the same software stack, or even just reusing the same relay with different hostnames, propagating NIP65 events for those relays requires more effort. Multiple relays could also be used to achieve better decentralization for a given use case that will be discussed later in Chapter 5. The most widely advertised software stack is unexpected: it is used by a single relay yet advertised by 40 127 users, which is a substantial share given that the NIP65 analysis identifies just over 140 000 users overall. This means, that this relay is advertised by over 25% of the users while only being a single relay *relay.momostr.pink*.

Looking at the 16th and 17th biggest relay in Table 4.6, we can see that there are relays that are advertised by a great number of users despite being invalid. First we assumed that this might be a mistake in the data collection or the NIP messages would only come from a single relay. However, after a deeper look into the data for the 16th relay we discovered that there are indeed over 4 000 distinct users (by public key) that are advertising the same invalid relay on over 90 relays. One of those relays is one of our own relays. In this case, the relay cannot be considered valid for most users, as it uses the `.local` domain, which is a reserved top-level domain [52] that is not routable on the public internet. For the 17th relay (`nostr.zbd.gg`), the

Rank	Relay	Valid	Users	detect_ratio
1	relay.momostr.pink	True	40 127	0.423 077
2	relay.damus.io	True	37 248	NaN
3	relay.primal.net	True	31 486	2.219 745
4	relay.mostr.pub	True	23 376	2.918 828
5	relay.ditto.pub	True	20 451	3.383 750
6	sendit.nosflare.com	True	18 432	NaN
7	nos.lol	True	17 000	NaN
8	relay.nostr.band	True	11 733	NaN
9	nostr.wine	True	10 723	NaN
10	nostr.sprovoost.nl	True	10 072	0.011 958
11	relay.snort.social	True	9 993	3.071 811
12	eden.nostr.land	True	6 564	NaN
13	nostr.mom	True	6 362	2.839 912
14	purplepag.es	True	5 361	NaN
15	nostr.bitcoiner.social	True	4 647	2.916 667
16	dwuainozfmexysuchxurpmclhdsovvjjybqfywxcidskmjbaad.local	False	4 081	NaN
17	nostr.zbd.gg	False	3 829	NaN
18	relayable.org	True	2 772	NaN
19	nostr.oxtr.dev	True	2 743	3.203 310
20	offchain.pub	True	2 671	2.783 088

Table 4.6: Top 20 Relay by Users

Software Rank	Relay Rank	Software	Users	Relays
1	1	git+https://github.com/hoytech/strfry.git	68 662	329
2	64	git+https://github.com/nanikamado/rockstr.git	40 127	1
3	2	https://git.sr.ht/ gheartsfield/nostr-rs-relay	21 838	223
4	50	https://gitlab.com/soapbox-pub/ditto-relay	20 461	2
5	16	https://github.com/Spl0itable/nosflare	18 595	8
6	27	https://github.com/v0l/memlay	9 994	3
7	13	NFDB	9 106	11
8	104	https://github.com/pablof7z/purplepag.es	5 361	1
9	5	git+https://github.com/Cameri/nostream.git	3 195	27
10	7	https://github.com/fiatjaf/pyramid	1 947	23
11	9	https://github.com/bitvora/wot-relay	1 737	19
12	8	git+https://github.com/cameri/nostream.git	1 433	23
13	6	https://github.com/fiatjaf/khatru	1 240	26
14	4	https://github.com/barrydeen/haven	840	28
15	3	https://github.com/bitvora/haven	757	54
16	31	custom	756	3
17	11	https://github.com/rnostr/rnostr	651	12
18	23	shugur	581	5
19	19	https://pkg.go.dev/fiatjaf.com/nostr/khatru	482	6
20	46	searchnos	434	2

Table 4.7: Top 20 Software by Users

reason for its invalidity is not immediately evident, as the domain appears syntactically valid. Also a static analysis of the domain name does not show any red flags, since the top level domain is a valid global domain. However, the domain name cannot be resolved. This is not a local issue that might be caused by the university DNS servers, since it cannot be resolved using Cloudflare (1.1.1.1) or Google (8.8.8.8) DNS servers either. This finding is another result that is an answer towards RQ3 and demonstrates that the found issues in the event conformity are not only present in single cases from single users, but rather can be observed on a larger scale what could be perceived as large scale non-random anomalies. It reinforces our finding, that unsupported message propagation is not an exception but rather a common occurrence.

4.4.1.1 Software Interaction

So far this propagated relays advertising topology displays only the usage of relays and software stacks by users, but does not show if there is any trend in usage and interaction in between software stacks. To get a insight into this question about interaction between software stacks, we display a metric that is the direct product of `NIP65` messages.

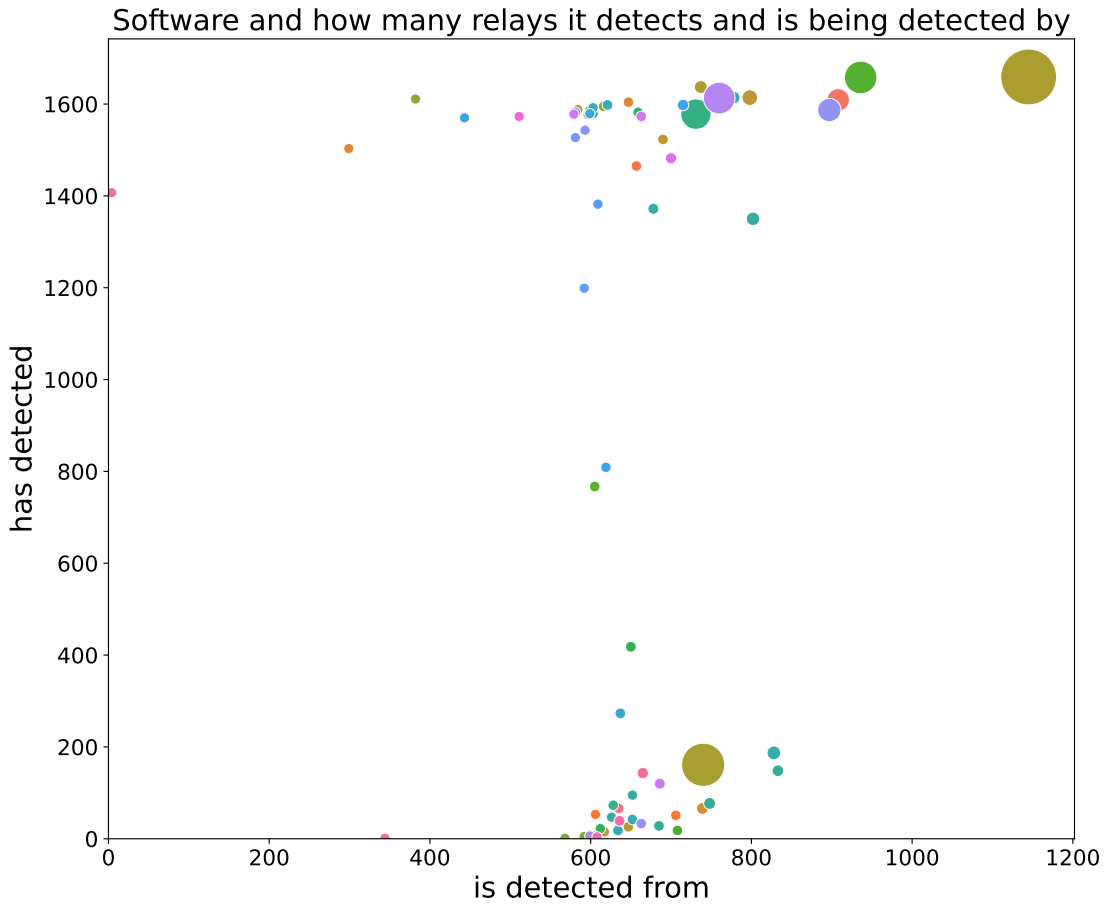


Figure 4.15: Software Relay Detection Vs Detected

We started by fetching all `NIP65` messages and adding the software stack for each relay where possible.

Since the messages do need to be stored on a relay, we can also fetch the software stack of the relay where the message is stored and thus get a metric indicating the discovery of software to software.

Figure 4.15 illustrates this metric. We can see for each given software stack how many times it was detected and how many times it allowed us to detect another software. We then continued to plot the amount of times a software stack was detected on the x-axis and the amount of times it allowed us to detect another software stack on the y-axis. The size of the bubble indicates the amount of users per software stack, which is the same metric as in Table 4.7. It is important to keep in mind that the software stack mentioned is not to be perceived as the programming language or framework used for the implementation, but rather the actual software stack as advertised in the `NIP11` document of the relay.

A relay being in the upper left corner indicates that it is a software stack that is not frequently detected from other software stacks, but when it is detected it allows us to detect a lot of other software stacks. An explanation for this could be possible interoperability issues with other software stacks. On the opposite, the lower right corner indicates software stacks that are frequently detected by other softwares stacks, but do not allow us to detect many others. If the amount of relays detected from such a software stack is zero or very low, a possiblity for this case are limitations, which could be defined in the `NIP11` document for each relay. Since each relay can announce limitations, such as maximal request size, maximal event size or payment required for usage, it is a possibility that our automatic discovery does not satisfy these limitations and thus we cannot detect other software stacks through the relay with the given software stack.

The indications of the required payments for events that propagate actual data such as kind 1 are clear. However, the limitations for payments as specified in `NIP11` are trivial. It states that "*payment_required: this relay requires payment before a new connection may perform any action*" [19] but does not specify any exceptions where this can be beneficial. One such exception could be that the relay allows certain event kinds to be published and fetched without payment, such as the `NIP65` events since they are beneficial for the discovery of relays. Such an exception would not only enable our research to get a better insight into the software ecosystem, but also allow for users to better discover each others and thus increase the degree of connectivity within the network.

Software stacks that are located somewhere in the middle where the amount of detections is similar to the amount of detected relays could be perceived as software stacks that are well connected within the network. However they probably do not display any greater interoperability and are good contributors to the overall connectivity of the network. The best example of such a software is the `hoyetech/strfry` [46] software stack, which is the most used software stack by relays and users and is located in the middle of the graph. With over 1 000 detected relays and a little bit over 800 times detected by others, it shows that is a greater discoverer than it is being discovered. The amount of detections being so high can be explained by the high adoption rate of the software stack in general. A good metric for the adoption rate and popularity of a software stack are the amount of github stars, which provides a good explanation for the high amount of detections.

Given the data that Figure 4.15 visualizes can be interpreted as an adjacency matrix, we can also interpret the network as a graph. For our cases such a graph could contain software stacks as nodes and the edges between them could be weighted by the count of the amount of times a software stacks have discovered each other. For a general usage one could calculate the distance between users by calculating their distance using the measurement of how many relays would be required to fetch information from to get from one user to another. However, the traversal of such a graph in the context of Nostr is not trivial, since we would need to build the graph for all users and relays, then allowing for the distance measurement using any algorithm of choice (such as Dijkstra's algorithm). A runtime search can be computationally expensive, as the lack of prior knowledge about the graph necessitates exhaustive exploration, incurring high bandwidth and request overhead without guaranteeing a result within a reasonable time frame. We therefore leave this

for future work.

4.4.1.2 Relay Interaction

In order to get a better understanding of the existing of such relays we did the same visualization as in Figure 4.15 across the relay dimension and received in Figure 4.16.

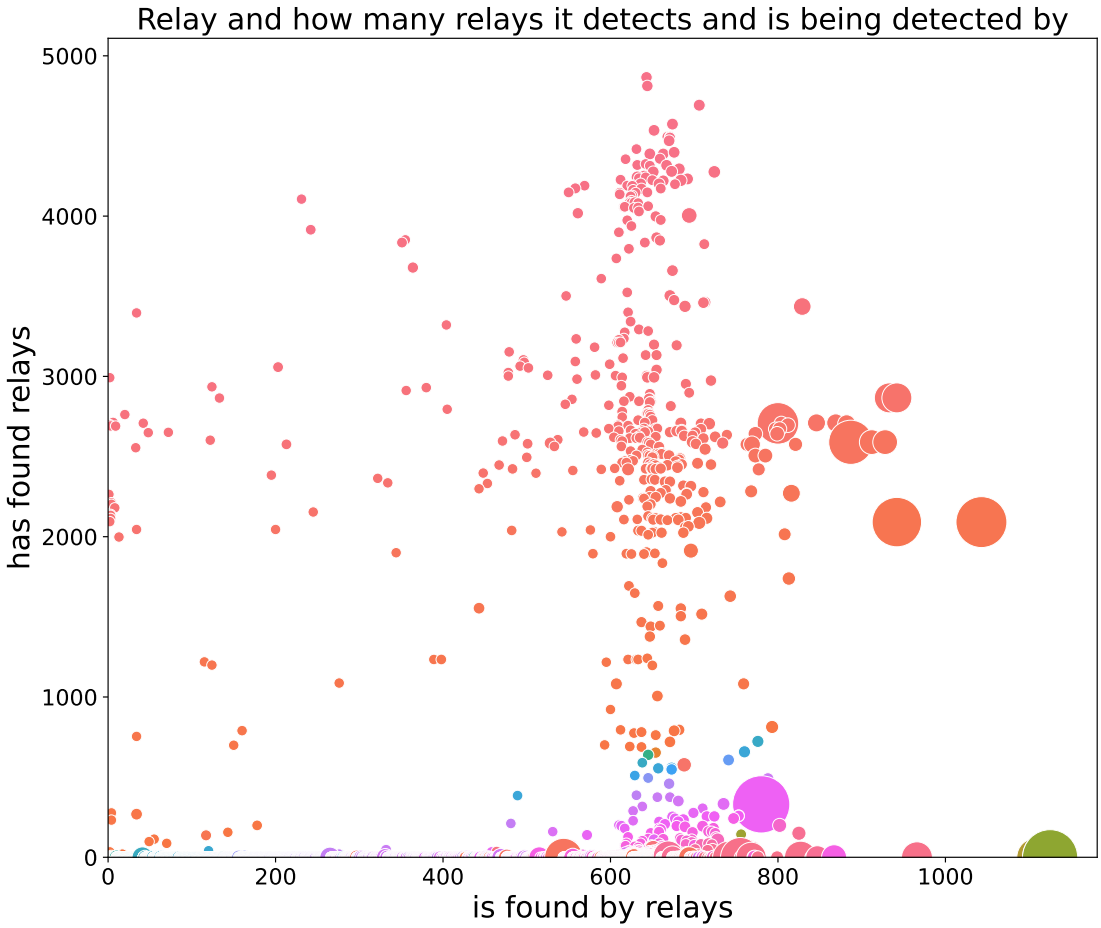


Figure 4.16: Relay Detection Vs Detected

The first thing to observe is the amount of relays located on the x-axis with a y-value (meaning they did not detect any other relays) is quite high. These are the relays that do not allow us to fetch any `NIP65` events from them. The reason can be as already stated previously due the requirement of payment for usage of the relay, that the relay is not reachable due to network related reasons or that they actually do not hold any `NIP65` messages. These relays would be interesting for future work since they might open the door for better understanding of the Nostr ecosystem from an economical perspective, since they might generate revenue. The fact that some of those relays are advertised by a great amount of people makes it likely that there is a non-negligible amount of revenue generated by those relays. But we see that in general a relay holds much more information about other relays than it is being discovered by other relays. These are the ones in the lower right corner of the Figure 4.16.

On the opposite side there is group of relays that are located in the upper left corner, meaning that they are the ones that are relays that do discover a lot of other relays, but are discovered by a lower amount of people. A good example of such a relay are our own two instances Homelab and UNIBE. We discovered that our relays are only advertised by a handful of users for being used, but do contain a lot of information about other relays. This is achieved by NIP65 messages being published to our relays by users that are not advertising our relays as being used by them. For our research perspective we appreciate people publishing NIP65 messages to our relays because it allows us to use it as bigger entry point for the discovery of other relays and the subsequent data collection. But also for the purpose of the network it is not a bad thing to have usages and relays advertisements being decoupled from only the exact relays that we are using. It spreads the information about other relays and users across the network which prevents the clustering of relays and users into groups that might not discover each other. Consequently, this spread of information is beneficial for the overall connectivity and is a good finding of relay usage patterns in the Nostr ecosystem.

4.4.2 Summary

For the last of our research questions we can summarize the results as follows.

Relay Discovery By Software RQ4.1 For relay discovery by software stack, we observe a skewed topology: many stacks contribute marginally, while a small number of stacks dominate the relay discovery of others. Stofware stack size also plays a role, as stacks with large user bases (but not necessarily many relays) tend to contribute little to discovery and are instead primarily discovered by other stacks. This makes sense if a user wants to limit interoperability issues since if one relay has problems, another one can directly take over without the danger of any issues. More insights into relay discovery could by possible by overcoming the paywalls of certain relays to get access to their NIP65 messages, which could be an interesting direction for future work. There is no clear correlation between the programming language of the software stack and the discovery of other software stacks. The programming languages itself do not show any clear trends, but rather show that there are software stacks in all major programming languages available.

User To Relay Usage Patterns RQ4.2 Analysis of users by software and relay topology reveals no direct relationship between the number of relays running a software stack and its user count. However, the largest software stacks by users are also among those with the highest number of relays. Meaning that a relay can be become popular and frequently used by users on a single relay (per software stack), but it is not required to have a lot of relays (for the used software) to get there. This is an indication that the amount of relays per software stack is not the key factor for the amount of users per software stack, but rather the amount of features and other factors outweigh the relay count. Other factors include applications running on top of the software stack such as Alexandria [53] or the general reputation of the software stack.

4.5 Further Findings

Throughout the analysis of our data model for the different research questions we have found some additional interesting findings that reside outside the scope of our RQs but are still interesting to be presented. The goal of the following section is to present these findings as additional information that might influence our discussion and conclusion in the later chapters.

4.5.1 IP-Address Analysis

During the execution of the artio-miner (cf. Section 3.2.2) to get to the data foundation to answer previous questions, we also took note of the DNS resolution result of the relays found through NIP65 events. This DNS resolution allows us in the first place to save resources during the execution of the artio-miner since we do not have to query and wait for timeouts from unreachable relays. As a second benefit, we are able to analyze which relays might have different public names pointing to the same infrastructure.

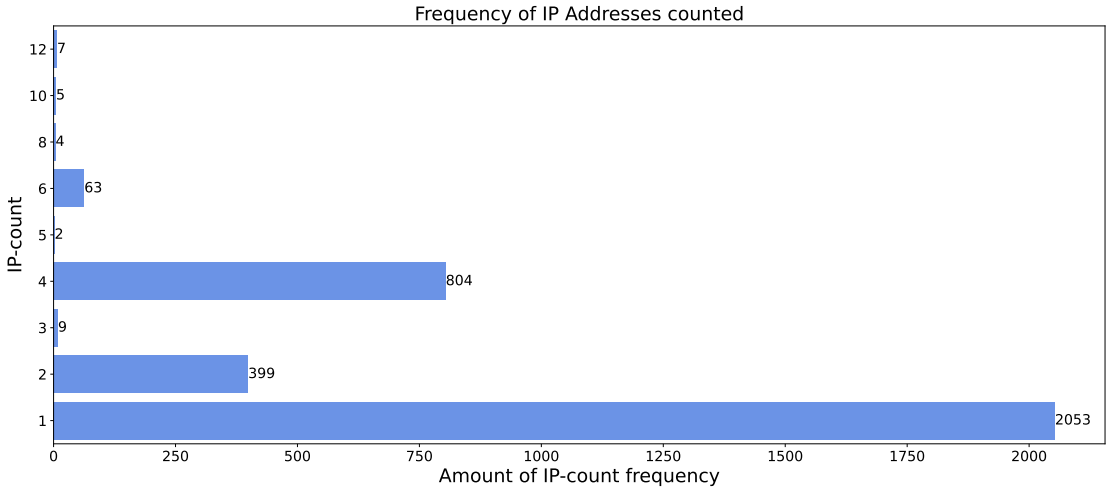


Figure 4.17: Amount Of IP Addresses Per Relay Counted During NIP65 Analysis

In Figure 4.17 we can see how many relays have how many IP addresses associated with them. The majority of relays have a single IP address associated with them, which is what we would expect from a typical relay setup. However there are some relays that have multiple IP addresses associated with them. The extreme case is a relay that has 12 different IP addresses associated with it. In conventional web infrastructure, associating multiple IP addresses with a single hostname is commonly used to achieve load balancing and high availability. In the context of Nostr relays, this is a notable observation, as such advanced configurations are not typically expected in a relatively young, community-driven ecosystem. However, our own development setup for `relay.artiostr.chrb`Homelab instance? demonstrates that this assumption does not necessarily hold. Although it resolves to multiple IP addresses (cf. Listing 4.2), the deployment is not highly available, but rather reflects a local or experimental (e.g., homelab) setup.

```
1 seg@artio:~$ dig relay.artiostr.ch +noall +answer
2 relay.artiostr.ch.      285     IN      A       172.67.220.56
3 relay.artiostr.ch.      285     IN      A       104.21.78.103
4 seg@artio:~$
```

Listing 4.2: DNS Record Lookup For relay.artiostr.ch

The fact that the domain name is resolved to two different addresses is due to the usage of Cloudflare [40] as a reverse proxy in front of our relay. The setup of the Homelab instance is not high available or load balanced and is running on a single instance. However, Cloudflare as a reverse proxy provider uses multiple IP addresses to provide redundancy and load balancing on their end. This shows that even though a relay might have multiple IP addresses associated with it, it does not necessarily mean that the relay is actually

running a highly available or load-balanced infrastructure.

On the other hand it is not only possible for one relay to have multiple IP addresses, but also for multiple relays to share the same IP address, but have a different hostname. A good such example for this can be seen in Figure 4.18.

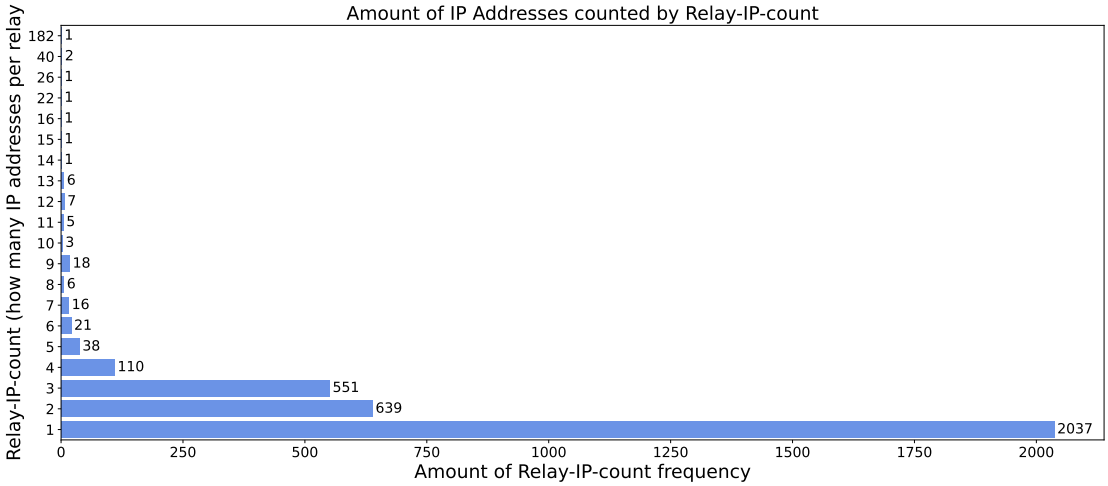


Figure 4.18: Amount Of IP Addresses Per Relay Counted

Whereas as expected the majority of IP addresses only have a single relay associated with them, there are some IP addresses that have multiple relays associated with them. Especially interesting is the case of the address that has 182 relays associated with them. Looking into our data we can find the IP addresses and their associated relays. The interesting fact is that both IP addresses share 182 relays that share the same domain and only differ in their subdomains. Using the same dig command as in Listing 4.2 we can check if there is a DNS wildcard record setup for this domain in Listing 4.3.

```
1 seg@artio:~$ dig *.nostr1.com +noall +answer
2 *.nostr1.com.           60      IN      A       154.38.180.90
3 *.nostr1.com.           60      IN      A       82.197.67.43
4 seg@artio:~$
```

Listing 4.3: DNS Record Lookup For *.nostr1.com

The wildcard DNS query returns two IP addresses, indicating that nostr1.com is configured with a wildcard record resolving to both. This suggests that multiple relay hostnames map to the same underlying infrastructure, rather than representing distinct relays. This can make it easier for clients to bypass censorship based on domain names, since blocking a single subdomain name would not be sufficient to block access to the relay infrastructure. However, since they are all using the same higher-level domain with the same IP addresses, it would be fairly easy to block access to the higher-level domain or the IP address directly.

4.5.2 IPv6 Adoption

Based on IP addresses collected during the NIP65 analysis, we also examined IPv6 adoption in the Nostr ecosystem. The amount of free IPv4 addresses is limited and the situation of obtaining new IPv4

addresses is a difficult. Since IPv6 would solve a lot of these problems, the topic of IPv6 adoption is interesting to analyze in the context of the Nostr network. Concerning IPv6 adoption, we observe that approximately 35% of all found IP addresses are IPv6 addresses, as shown in Figure 4.19.

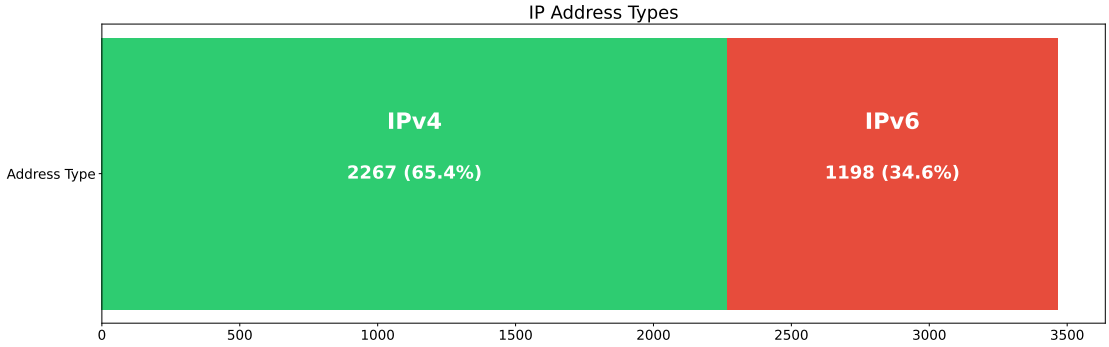


Figure 4.19: IP Address Type Distribution (IPv4 Vs IPv6)

This is a fairly high number for a communication protocol still in development, given that the global IPv6 adoption rate is around 45% as of January 2026 according to Google [54].

5 Discussion

This chapter discusses the present results and reflects on the implications of these results in the context of our research questions and the Nostr ecosystem as a whole.

5.1 Relays And Users

First of all, we want to discuss topics around the relays and users as the main entities in the Nostr ecosystem, for which we faced some challenges during our research process as we have already seen in the Chapter 4.

5.1.1 Identification Of Distinct Entities

The identification of distinct entities within the Nostr network has presented us with several challenges, as evidenced by the findings presented in Chapter 4. Key entities in the protocol ecosystem include relays and users, identifiable respectively through relay `hostnames` and pubkeys (for both end-users and relay operators).

The challenge of entity disambiguation manifests in two primary dimensions. On the one hand, there are relays that share the same IP address but have different hostnames and might be the same relay infrastructure in the background. On the other hand, there are also valid reasons for a single IP address to host multiple distinct relays. Common examples of such architectures include deployment patterns utilizing reverse proxies or content delivery networks (CDNs) [55]. This is exemplified by our own infrastructure: the relay at relay.artiostr.ch operates behind Cloudflare as a reverse proxy, rendering it impossible to definitively determine whether multiple hostname entries represent the same underlying relay infrastructure or independent instances.

Another challenge exists at the user level, where it is impossible to determine if two pubkeys belong to the same user or not. The only possibility would be to correlate the activity of two pubkeys together with other dimensions such as the content of published events and the used ip addresses. Unfortunately, this is

something that is not easily achievable and is highly prone to producing errors, as users might have similar interests and might be using the same public IP address at some point in time.

This question is important in the context of the finding in Section 4.5.1 where we concluded that there are multiple hostnames pointing to the same IP address and they are all using the same parent domain *nostr1.com*. As we have seen in the results section we are certain that there is a wildcard DNS entry for the nostr1.com domain, and they only differ in some part of the NIP11 such as their operator pubkey.

Since they are all using the same software stack and they require payment for usage, we can only make a certain statement that there are differences in the NIP11 documents of these relays. As discussed earlier, NIP11 documents can be manipulated to create the appearance of distinct relays, although the underlying infrastructure and data storage remain identical.

5.1.2 Undiscoverable And Unreachable Relays

Coming from the perspective of the recognition of distinct entities, the question of undiscoverable and unreachable relays is also an interesting one. It was another significant challenge we faced during our research: there are a lot of relays that are not discoverable or unreachable from our servers due to multiple reasons. Whereas we do not have knowledge about relays that are not discoverable at all, we have a good understanding of the reasons why some relays are unreachable and how this can impact our research and the Nostr ecosystem as a whole. The most prominent reason we have identified is the problem of hostnames not being properly resolved to IP addresses by our DNS resolvers. While it is possible that some of the hostnames would be able to be resolved by other public facing DNS resolvers, the biggest part of their unresolvable hostnames is due to malformed or misspelled hostnames. The biggest example of such a hostname we have already seen in Section 4.4.1 with the *.local* domain.

The question on how to differentiate between a relay that is actually invalid such as *127.0.0.1* as stated in RFC 5735 [44] and one where there might just be a typo in the hostname was a challenge. For the scope of our research we differentiated between the case where a hostname is clearly invalid, such as invalid domains (e.g. no point contained) and ones that could be valid. For the ones that are clearly invalid we did not even try to fetch information from them, since it could have produced invalid data and skewed our results. For the other ones, we treated them as valid for the moment and tried to resolve them and see if we can get any IP address from them. If we cannot fetch any IP address from them, we treated them as unreachable and did not include them in our analysis of the IP addresses. Any follow-up errors during the fetching process were then treated within the implementation of the corresponding component and taken into account for the final data production. However, this does not prevent any invalid relays being propagated in the network as we have already seen and will be discussed in Section 5.3.

5.2 Applications Using Nostr

The nostr1.com domain serves not only as an example for the challenge of identifying distinct entities, but also illustrates applications built on top of Nostr as an underlying communication layer. When introducing Nostr to unfamiliar audiences, it is often described as a "decentralized social media network" to aid intuition. However, this simplified comparison is misleading when compared to the protocol's own definition which does not mention social media or any specific application:

> *"An open social protocol with a chance of working.*
> *Like the internet itself: open and chaotic." [3]*

During the research process, we have seen that many NIPs are heavily focused on the social media use case (e.g. direct messages, reactions, etc.), and we thought about other possible use cases for the protocol. The

nostr1.com domain and the corresponding relays are an actual example of a non-social media application that is built upon the Nostr protocol. When looking at the page rendered when opening one of those relays in a web browser, we get an explanation that the relays are used for the alexandria client and moderated by the members of gitcitadel [56]. The alexandria client states to be a reader and writer for curated articles, wiki docs and other content [53]. This seemingly formal definition of the application is a bit misleading since it actually seems to be a an e-book reader and writer platform used for sharing documents. Since sharing e-books and documents is something that might violate copyright laws in some countries, it is possible that the operators of the relays are using the distributed and decentralized nature to their advantage to make it harder to take down the content that is shared on their relays. However, the legality of content and applications shared across the Nostr network is outside the scope of this work.

Beyond this, the protocol also supports use cases that leverage its advantages without raising legal concerns. Looking at latest trends in the software development and architectures, it is comprehensible that microservices and distributed systems are becoming more common within cloud-native applications such as Kubernetes being widely adopted. In such distributed systems there is a need for a communication layer between the different components and services. A highly popular solution for this problem is to use a message bus for distributed event streaming with Apache Kafka [57] being one of the most popular and widely adopted solutions. Given the strictly event-based pattern of the Nostr protocol, it could be possible to use it as such a message bus for distributed event streaming in a microservices architecture. There is no actual need for any of the social media related NIPs in such a use case, and one could reduce the protocols' features to its core and `NIP01` for simple event streaming. Since for Kubernetes environments the high availability and scalability of the event streaming is crucial, Nostr is a good candidate since we can easily deploy multiple relays and use them as distributed event storages. One challenge that presents itself in such a use case is the fact that the storage requirement for the events can be quite high and comes with additional challenges when managing Persistent Volume Claims in Kubernetes.

Another advantage of `NIP01` is the simplicity of the client component, enabling lightweight implementations when more complex NIPs are excluded. These facets are especially interesting in the context of IoT devices and message brokering. A direct comparison of Nostr with MQTT [58] and Apache Kafka [57] in the context of microservices and IoT applications would be an interesting topic for future research. The possibility of developing alternative implementations of the protocol that are optimized for such use cases could open up new opportunities. Especially for the IoT use case where the devices are often resource constrained (e.g. power, CPU, memory and more) and have limited network connectivity, a lightweight and decentralized protocol such as Nostr could be a good fit for the communication layer since it does not require a central server and can handle intermittent connectivity. Since the relays store the events for a certain period of time, it can also handle the situation where a client would like to query for events that were published while the client was offline. A comparison between existing protocols for message brokering and event streaming in the context of microservices and IoT applications and Nostr would be an interesting topic for future research.

The paradigm in general is that we have a decentralized and distributed event streaming platform that can be used for a wide range of applications, not only social media related ones with the possibility to adapt the relays with custom components to other protocols and standards. This paradigm combined with the lightweight core protocol is what we believe is the value that one could leverage for other software development and architectural use cases, and it is something that we would like to further explore in the future as already mentioned in Section 5.6.4.

5.3 Interesting Violations

As we have seen in the results section, there exist interesting violations of the Nostr protocol that we have identified during our research process. Whereas some of the mistakes are clearly unintentional and can be easily explained by human error, such as typos in the hostnames, there are some other cases that might be intentional.

During the analysis of the usage topology in order to answer RQ4 we have identified a relay with the hostname `dw6uain7ozfm7exys7uchxurpmclhdsovvjjybqfywx6cidskmjb7aad.local` that is advertised by multiple users. There are 4 081 users that do advertise this relay in their NIP11 documents, but the hostname is not resolvable and the relay is not reachable, since the domain is a *.local* domain. The specific rules for the *.local* domain are defined in RFC 6762 [52] and they state that the *.local* domain is reserved for use in local area networks and should not be used in the global DNS system. Initially, we assumed that the relay was advertised by one or a few users via their NIP11 documents. Further analysis shows that it is advertised across more than 60 different relays, making a simple misconfiguration unlikely. This data indicates a large-scale protocol violation which, however, does not affect the core functionality of the Nostr ecosystem. It speaks towards the fact that the protocol is flexible and error tolerant and can deal with also bigger scale violations. It also proves our previous statement that any user can easily create a new pubkey and spread information across the network without any possibility to easily identify them, since the simple EVENT publishing act with NIP65 is handling the message as expected.

Another interesting violation is the case of the EVENT types *NEG-OPEN* and *NEG-ClOSE*. As we see in our example NIP11 document in Listing 2.3 we do not advertise NIP77. Nevertheless, we observed traffic related to NIP77 on our relays, constituting a clear violation. Although we did not further investigate the required implementation, the presence of such traffic indicates that clients or relays attempt to use this NIP and publish corresponding events. A possible explanation is they are testing if the necessary implementation for NIP77 is present on the relay although not advertised. One possible explanation is that clients probe for NIP77support, even though it is not advertised. This behavior aligns with the experimental and permissive nature of Nostr, which is described as *"a protocol with a chance of working"* [3]. However, it was not a single day where we did see the activity but rather a longer period of time, which makes it more interesting. It might be an indication that some client is actually trying to use NIP77 and publishing events related to it without any success since the relay does not support it.

5.4 NIP-11 Adoption

One recurring topic during all our research and also the previous discussion points is the question of the NIP11 adoption and the published NIPs within the NIP11 document. As we have seen in Chapter 4, there are many relays that do not publish any NIP11 document at all. Even when they do (as we are certain of for our own relay) there are certain clients that do not respect the NIP11 document and try to use NIPs that are not advertised by the relay. While this might be fine if the user is knowingly accepting the risk of improper handling of the events, most users are not directly interacting with the relays but using a client. None of the clients that we have looked at or personally used during our research process have any kind of warning or indication for the user that they are trying to use a NIP that is not supported by the relay.

In some cases one can specify for which type of publication or data fetching a relay should be used, as it is the case for amethyst. However, in order to be able to configure the client in such a way, the user needs to have a solid understanding of the Nostr ecosystem and what the NIPs are actually doing in order to understand the implications of using a relay for each specific purpose. This is a problem given Nostr aims to be open and accessible for everyone. Even those who do not have a technical background and will lead to the situation where the user is relying on the client to handle the NIP11 document and the NIPs

advertised by the relay in a proper way. For the more technical users this might not be a problem, but for the less technical users this might lead to a situation where they are using relays that do not support the NIPs that they are trying to use and therefore not getting the expected behavior from the protocol.

It might even lead to more severe situations, when a client is not properly checking if a relay supports encrypted direct messages. When the user is trying to use this feature on a relay that does not support it, it might lead to the situation where the user is sending unencrypted direct messages without even realizing it. One can ask themselves when it has been the last time that they have read the full source code of an application that they are using and if the source is even open source available to be able to do so. This is a problem that is not only present in the Nostr ecosystem but also in the software ecosystem as a whole, where users are relying on the software to handle certain functionalities and features without having a good understanding of how they work and what the implications are. The exact same challenge is present in other CDV ecosystems, such as the Bitcoin ecosystem.

However, to get back to the Nostr ecosystem, during our research we have wished multiple times that the `NIP11` document would be mandatory and must be respected by the clients and other relays. In order to be able to make sure that the users are getting the expected behavior from the protocol, we need to avoid the situations, where users are using relays that do not support the NIPs that they are trying to use. It would heavily improve the user experience and the overall ecosystem if there would be a better adoption of the `NIP11` document and a better handling of it by the clients and other relays. On the other hand, wherever there is a mandatory requirement, there is also no long time until there is someone trying to bypass said limitations. For the protocol this would mean, that one could just publish everything in the content field and specify the event kind 1. Whereas this would work for the user it would work against the spirit of the protocol where all we have certain feature and we are capable of distinguishing them based on the kind. The point that we have left out for the moment is the fact that without said violations in the protocol this thesis would not have been as extensive as it is right now.

As future work, we propose improving adherence to the `NIP11` document and the NIPs declared within it, without imposing it as a mandatory requirement. One possible starting point for this would be to analyze how clients respect any limitations specified in the `NIP11` document and record if there are any patterns in the impact of said limitations on the user experience and the overall ecosystem.

5.5 Artio as End-to-End Analysis Platform

Lastly for the discussion we would like to reflect on the topic of the platform and architecture of Artio and what implication this contribution does have for the research process in general. We would like to highlight, that this platform is not only a contribution for the Nostr ecosystem, but also a contribution for the research process in general, since it can be used as a starting point for future research on the Nostr protocol and ecosystem. At the beginning of our research, we did not expect the platform to grow into a large and complex system.

The main goal was to separate the research process from the implementation of the platform and to have a flexible and extensible platform. Whereas the artio-relay is a strict Nostr related component of Artio, all other components are not directly related to the Nostr protocol. One can argue that the insight component is also directly related, where we partially agree. However, only the content of the insights and the structure of the schemas are Artio and Nostr specific. The rest of the platform is not directly related to the Nostr protocol, and the main idea of the go-db-etl [7] and other components is to be reusable for other research projects since its scaling has proven to be highly effective and efficient for our purposes. We do not see any reason why someone else might not be able to use it for their own research projects.

A disadvantages of the Artio platform is that is requires advanced technical expertise to deploy and operate.

Software development skills alone are insufficient, as reliable operation also depends on solid infrastructure management. Throughout our study, we encountered issues such as storage exhaustion, misconfigured log rotation, and network instability, which, while not directly related to the research, significantly affected the process. The setup itself is something that one could think about detaching from docker containers. For us it had the purpose to be really easy to get going in the beginning and it allowed us to quickly iterate and deploy new components without having to worry about dependency nightmares and other issues that can arise when setting up a complex platform.

In the long term, however, several operational issues have already become apparent. A representative example is the use of `PostgreSQL` [28]instances deployed as Docker containers. While containerization simplifies networking (e.g., port mappings), it complicates version upgrades. Some instances still run on `PostgreSQL` 15, which is approaching obsolescence relative to current releases (e.g., `PostgreSQL` 18). Upgrading containerized databases typically requires full data export and reinitialization, leading to substantial storage overhead and service downtime. This introduces non-trivial operational challenges that are undesirable in a research infrastructure.

From the monitoring aspect we were also facing some challenges with the fact that the Prometheus Node Exporter running in a docker container presented some challenges with the monitoring of the host system, since it is not running directly on the host but in a container. We were able to work around this issue by using mappings for the proc and sys directories, but it is still not ideal and can lead to some issues with the monitoring of the host system. The full stack of monitoring could also be expanded with some long term collection tools such as splunk or similar, which would allow for some deeper insights into the performance and IP addresses from the `HAProxy` [59], but this data would then need to be correlated with the data from the relay itself, which could lead to some challenges in the data analysis process. Over all the monitoring did fulfill its purpose and allowed us to efficiently identify and react to any issues with the platform, but it is still something that could be improved in the future.

5.6 Future Work

While we have already mentioned some ideas for future work in the previous Chapter 5 and the Chapter 4, we would like to summarize and further elaborate on some these ideas in this section and also mention some other ideas for future work that we have not mentioned before.

5.6.1 Platform and architecture of Artio

As already stated in the previous Section 5.5 we would like to further develop and improve Artio in the future. Running after the personal moto of *"if it works it ain't stupid"* the platform did fulfill its purpose and allowed us to successfully conduct our research and answer our research questions, but there are still some areas where we see potential for improvement.

However, the strong coupling between the platform and Nostr should be reduced in future work, as it may limit adoption by researchers whose work is not directly related to the Nostr protocol. Given this detachment, it would also allow us to publish the artio-orchestration [39] component together with the other components as a baseline which could open the door for some proof-of-concept projects that are not directly related to Nostr and would highlight the flexibility and extensibility of the platform. The monitoring and alerting stack is also something that could be improved during this detachment process. It is currently heavily focused on the Nostr related components and metrics, but it could be expanded to include more generic metrics and alerts that are not directly related to Nostr, which would still be useful for the overall health and performance of the platform.

5.6.2 Violations analysis

As we have seen in Section 5.3, there are some interesting violations of the Nostr protocol that we have identified during our research process, but we have not further looked into them and analyzed them in depth. Naturally this is something that could be a topic for future research, since it would allow us to better understand the implications of these violations and how they impact the overall ecosystem. The baseline of data for this analysis is already present the data we have collected during the process of this work.

A promisiong new feature that arised during the time of research is the possibility of extending the search functionality (NIP50). This would allow for distributed search across multiple relays, which could be a good starting point for this analysis. It would allow us to search for specific events related to the violations and see how they are distributed across the network and if there are any patterns in their distribution. Furthermore, given the existing traffic on the relays and the fact that we have established the functionality over time, one could use the existing relay implementation to try to extend its functionality or branch it into its own new implementation.

5.6.3 Extension and Enrichment of the Research

The possibilities for the extension and enrichment of the conducted research are manifold. Some more simple ideas would include the analysis of events stored on the relays that do require payment for usage, such as the ones on the nostr1.com relays, to see if there are any patterns in the content of these events and if there are any implications for the users that are using these relays. The effect of just paying the fee for using the relay and not actually publishing any events or fetching any data from it could also be an interesting topic for research, since we could compare the effect of paywalled relays into our discovery and usage topology with the effect of free relays and see if there are any differences in the patterns of usage and discovery for these two types of relays.

A more social approach on conducting interviews and speaking with actual developers and users of the Nostr ecosystem could lead to its own social research project, where we could try to understand the motivations and reasons behind the usage of certain relays and the implications of the violations for the users and developers in the ecosystem.

Further more it might lead to some relay owners letting as use there own relays for testing and research purposes, which could extend the possibilities for the research even further.

5.6.4 CDV in Nostr

For future research directly based on the gained insights from this thesis, we could extend to probe and investigate the specific communications between relays and clients. By testing pair-wise relay-relay or relay-client communication that use different software stacks, we could try to identify if there are any differences in the way they handle the events and if there are any implications for the users and developers in the ecosystem. One would get a better understanding of the effective features for a software stack which would generate insights into the implementations spectrum and what key feature drive interoperability and compatibility in the ecosystem. Furthermore, we could also try to implement an automated differential analysis of ‘effective features’ of a relay vs. its ‘claimed features’ as specified in the NIP11 document.

Apart from the specific violations and where to extend them into future research, what remains to be investigated is the question of how and if the paradigm of CDV [1, 2] can be leveraged into more conventional protocols and ecosystems. One could try to implement to extract the main ideas of Nostr (especially the slim and lightweight core protocol) and apply it to a more conventional software system such as any event driven BSS or a microservices architecture and see if the concept of CDV can be applied to such a system and if it can bring any benefits in terms of flexibility, scalability and resilience Since this

most probably would require a significant amount of development work, it could be a good starting point to try to implement a proof-of-concept for such a system and see if the concept even proofs to be useful in such a context before investing even more resources. Since such a test would require a baseline existing software system to be able to compare the results with, and the financial impact due to the initial required development work, it would be good to try to find a partner in the industry that would be interested in such a test and could provide us with the necessary resources and support to conduct it.

The biggest challenge for such a test would be the financial impact that someone needs to invest just relying on the fact that the concept of CDV can bring some benefits in terms of flexibility, scalability and resilience, which is something that is not guaranteed and would only be answered after the cost-intensive test has been conducted.

6 Conclusion

Coming to the end of this thesis, we can summarize the main findings and contributions of our research. Overall, the contribution on a methodical level is that we built, deployed, and successfully operated a large-scale Nostr analysis platform for investigation and answering of our research questions. On the application level, the contribution is present in the form of the artio-relay [23], which is a custom relay implementation that we developed for participating in the Nostr network and collecting data for our research. This methodical and applicatory contribution has then been used to actively participate in the Nostr network and analyze the ecosystem in the scope of our research questions about software usage and the ecosystem itself. The result overall is a large-scale investigation of software variability in Nostr and its manifestation in the wild. The platform itself is capable of being used for further research in the Nostr ecosystem and can be easily adapted to other use cases and research questions, which is something that we envision for future work.

Within the scope of this study and the collected data, we addressed four research questions. We were able to determine within RQ1 that the ecosystem is much more diverse than we initially expected, with the presence of a large number of different software implementations and relays than we had anticipated. The combination of supported NIPs is also something that we were able to highlight and group across the discovered landscape. We found that some NIPs are much more likely to be used together than others, which we expected but are now able to back up with data and analysis. Unexpectedly, the number of relay owners is much greater and more diverse than expected, keeping in mind that the barrier for creating new pubkeys is really low.

The event landscape RQ2 is much more diverse than expected and displayed an unexpectedly high amount of non-conformant events that should not be supported by our relays, which led to the discovery of some interesting patterns within the NIPs and how users are using them in the wild. Some NIPs require greater trust in the relays than others, where this trust is probably highly reliant on the `NIP11` document published by the relays.

This manifestation of allegedly non-conformant events is something that we then carried into RQ3, where we were able to see that a majority of events that are published to our relays are actually not conform to

the NIP11 document published by the relays. This is something that we expected but were able to back up with data and analysis. For usage trends, we were also able to gather data and highlight some interesting findings. The fact that more relays does not necessarily mean more users per relay is not something that we expected but discovered during the data analysis process.

For RQ4 we found that the usage topology of the Nostr ecosystem is much more complex than expected, with a large number of different relays and users that are interconnected in a complex way. Some relays are much more popular than others, which is something that we expected but are now able to back up with data and analysis. For the software usage topology, we were able to find that the amount of relays used is not necessarily correlated with the amount of users, which we did not expect but discovered during the data analysis process.

Overall, we have proved that Nostr is not only a protocol statically present in theory within its GitHub repository [13], but it is actually a living ecosystem with a large number of different software implementations, relays, and users that are actively using the protocol and its features in the wild. The results empirically validate prior CDV work [1, 2], showing that it is not merely a theoretical construct but observable in practice. Moreover, they demonstrate that CDV can be systematically analyzed using appropriate tools and methods, as realized through our research process and the *Artio* platform..

Personal Thoughts

Overall, I am happy with the results of the research process and the contribution that we were able to make with our research. Comparing the insights we had at the start of this project, where we had to rely on singular points of information from websites such as nostr.com [3] and nostr.watch[60], to the insights that we have now after collecting and analyzing a large amount of data from the Nostr ecosystem, we can see that our understanding of the ecosystem has significantly improved. We are now able to back up our insights with data and analysis instead of just relying on singular points of information. It has been an interesting and rewarding process that over time did not fail to surprise and challenge our own skills, as well as the software and hardware used to investigate the ecosystem. The combination of skills in multiple domains such as software development, systems engineering, data analysis, and more was something that was pleasantly challenging and highly rewarding.

Acknowledgments

Lastly, I would like to thank all the people that supported me during the course of this thesis, especially my supervisor Roman Bögli for his support, guidance, and inspiration during the research process. For the uncomplicated and quick support with the University server, I give big thanks to Peppo Brambilla for his help. Last but not least, I would like to thank all the people that are actively participating in the Nostr protocol and any other open-source project, since they are the ones that make such research possible by spending their time and energy contributing to open-source projects.

List of Figures

Listings

List of Tables

Bibliography

[1] R. Bögli, A. Boll, A. Schultheiß, and T. Kehrer, "Beyond Software Families: Community-Driven Variability," in *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering, FSE Companion 2025, Clarion Hotel Trondheim, Trondheim, Norway, June 23-28, 2025*, L. Montecchi, J. Li, D. Poshyvanyk, and D. Zhang, Eds., ACM, 2025, pp. 571–575. DOI: 10.1145/3696630.3728501 (cit. on pp. 1, 4, 7, 10, 27, 58, 61).

[2] R. Bögli, A. Boll, A. Schultheiß, and T. Kehrer, "Community-driven variability: Characterizing a new software variability paradigm," *Automated Software Engineering*, vol. 33, no. 2, p. 67, Dec. 2026. DOI: 10.1007/s10515-026-00594-0 (cit. on pp. 1, 4, 7, 10, 25, 27, 58, 61).

[3] fiatjaf. "Nostr.com," Accessed: Jan. 14, 2026. [Online]. Available: https://nostr.com (cit. on pp. 1, 4, 5, 7, 8, 53, 55, 61).

[4] R. Bögli, "Towards Systematic Treatment of Community-Driven Variability," in *International Conference on Software Engineering, ICSE - Companion Proceedings*, IEEE, 2026. DOI: 10.1145/3774748.3787644 (cit. on pp. 4, 10).

[5] K. Pohl, G. Böckle, and F. Van Der Linden, *Software Product Line Engineering*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005. DOI: 10.1007/3-540-28901-1 (cit. on p. 4).

[6] S. Apel, D. Batory, C. Kästner, and G. Saake, *Feature-Oriented Software Product Lines: Concepts and Implementation*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013. DOI: 10.1007/978-3-642-37521-7 (cit. on pp. 4, 7, 13).

[7] M. Kaiser, *Ooemperor/go-db-etl*, Dec. 15, 2025. Accessed: Jan. 14, 2026. [Online]. Available: https://github.com/ooemperor/go-db-etl (cit. on pp. 5, 16, 22, 56).

[8] Y. Wei and G. Tyson, *An Empirical Analysis of the Nostr Social Network: Decentralization, Availability, and Replication Overhead*, version 2, 2024. DOI: 10.48550/ARXIV.2402.05709 pre-published (cit. on p. 5).

[9] U. Jeong, L. H. X. Ng, K. M. Carley, and H. Liu, "Navigating Decentralized Online Social Networks: An Overview of Technical and Societal Challenges in Architectural Choices," 2025. DOI: 10.48550/ARXIV.2504.00071 pre-published (cit. on p. 5).

[10] H. Kimura, R. Ito, K. Minematsu, S. Shiraki, and T. Isobe, "Not in The Prophecies: Practical Attacks on Nostr," in *2025 IEEE 10th European Symposium on Security and Privacy (EuroS&P)*, Venice, Italy: IEEE, Jun. 30, 2025, pp. 585–606. DOI: 10.1109/EuroSP63326.2025.00040 (cit. on p. 5).

[11] T. Kehrer, T. Thüm, A. Schultheiß, and P. M. Bittner, "Bridging the gap between clone-and-own and software product lines," in *ICSE*, Piscataway: IEEE, May 2021, pp. 21–25, ISBN: 978-1-6654-0140-1 (cit. on pp. 7, 13, 27).

[12] K. Czarnecki, U. W. Eisenecker, and U. Eisenecker, *Generative Programming: Methods, Tools, and Applications*, 6. print. Boston Munich: Addison Wesley, 2005, 832 pp., ISBN: 978-0-201-30977-5 (cit. on pp. 7, 13).

[13] nostr-protocol. "Nostr-protocol/nips: Nostr Implementation Possibilities," Accessed: Jan. 14, 2026. [Online]. Available: https://github.com/nostr-protocol/nips (cit. on pp. 8, 10, 13, 14, 41, 61).

[14] "Explore Nostr Apps," Accessed: Jan. 14, 2026. [Online]. Available: https://nostrapps.com/ (cit. on p. 8).

[15] "Damus," Accessed: Feb. 4, 2026. [Online]. Available: https://damus.io (cit. on p. 8).

[16] "Primal," Accessed: Feb. 4, 2026. [Online]. Available: https://primal.net/home (cit. on p. 8).

[17] fiatjaf_, *Fiatjaf/relayer*, Jan. 10, 2026. Accessed: Jan. 14, 2026. [Online]. Available: https://github.com/fiatjaf/relayer (cit. on pp. 8, 14).

[18] nostr-protocol. "Nips/01.md at master · nostr-protocol/nips," GitHub, Accessed: Jan. 27, 2026. [Online]. Available: https://github.com/nostr-protocol/nips/blob/master/01.md (cit. on pp. 9, 74–83).

[19] nostr-protocol. "Nips/11.md at master · nostr-protocol/nips," Accessed: Jan. 14, 2026. [Online]. Available: https://github.com/nostr-protocol/nips/blob/master/11.md (cit. on pp. 9, 15, 46).

[20] Software Engineering Group, Institute of Computer Science, University of Bern. "Home," Software Engineering Group, Accessed: Jan. 14, 2026. [Online]. Available: https://seg.inf.unibe.ch/ (cit. on p. 10).

[21] M. Kaiser, *SEG-UNIBE/artio*, SEG UNIBE, Mar. 3, 2026. DOI: 10.5281/zenodo.18849681 Accessed: Mar. 3, 2026. [Online]. Available: https://github.com/SEG-UNIBE/artio (cit. on pp. 10, 14).

[22] M. Kaiser, "Towards Large-Scale Variability Investigations in Nostr," Gesellschaft für Informatik, Bonn, 2026. DOI: 10.18420/SE2026-WS_31 (cit. on pp. 10, 14).

[23] M. Kaiser, *SEG-UNIBE/artio-relay*, SEG UNIBE, Jan. 12, 2026. Accessed: Jan. 14, 2026. [Online]. Available: https://github.com/SEG-UNIBE/artio-relay (cit. on pp. 11, 14, 16, 17, 33, 36, 60).

[24] Nostrocket. "Nostr Event Kinds Reference," Accessed: Jan. 14, 2026. [Online]. Available: https://nostr.dev/ (cit. on pp. 13, 33).

[25] Prometheus Authors. "Prometheus - Monitoring system & time series database," Accessed: Jan. 14, 2026. [Online]. Available: https://prometheus.io/ (cit. on pp. 14, 17, 34).

[26] M. Kaiser. "Grafana," Grafana Artio, Accessed: Apr. 7, 2026. [Online]. Available: https://grafana.artio.inf.unibe.ch/public-dashboards/b739854e147747bba285bfd3bd89e521 (cit. on pp. 14, 34).

[27] "The Go Programming Language," Accessed: Jan. 14, 2026. [Online]. Available: https://go.dev/ (cit. on pp. 15, 16).

[28] PostgreSQL Global Development Group. "PostgreSQL," PostgreSQL, Accessed: Jan. 14, 2026. [Online]. Available: https://www.postgresql.org/ (cit. on pp. 15, 17, 57).

[29] Docker Inc. "Docker: Accelerated Container Application Development," Accessed: Jan. 14, 2026. [Online]. Available: https://www.docker.com/ (cit. on p. 15).

[30] M. Kaiser, *SEG-UNIBE/artio-miner*, SEG UNIBE, Jan. 27, 2026. Accessed: Jan. 27, 2026. [Online]. Available: https://github.com/SEG-UNIBE/artio-miner (cit. on pp. 15, 16, 43).

[31] Neo4j, Inc. "Neo4j Graph Database & Analytics – The Leader in Graph Databases," Graph Database & Analytics, Accessed: Jan. 14, 2026. [Online]. Available: https://neo4j.com/ (cit. on pp. 15, 17).

[32] nostr-protocol. "Nips/65.md at master · nostr-protocol/nips," GitHub, Accessed: Jan. 14, 2026. [Online]. Available: https://github.com/nostr-protocol/nips/blob/master/65.md (cit. on p. 15).

[33] D. Linstedt and M. Olschimke, *Building a Scalable Data Warehouse with Data Vault 2.0*. Boston: Morgan Kaufmann, 2016, ISBN: 978-0-12-802510-9 (cit. on p. 16).

[34] M. Kaiser, *SEG-UNIBE/artio-insight*, SEG UNIBE, Jan. 9, 2026. Accessed: Jan. 14, 2026. [Online]. Available: https://github.com/SEG-UNIBE/artio-insight (cit. on pp. 17, 22).

[35] Grafana Labs. "Grafana — Query, visualize, alerting observability platform," Grafana Labs, Accessed: Jan. 14, 2026. [Online]. Available: https://grafana.com/grafana/ (cit. on pp. 17, 34).

[36] "Matplotlib — Visualization with Python," Accessed: Mar. 31, 2026. [Online]. Available: https://matplotlib.org/ (cit. on p. 19).

[37] M. Waskom, "Seaborn: Statistical data visualization," *Journal of Open Source Software*, vol. 6, no. 60, p. 3021, Apr. 6, 2021, ISSN: 2475-9066. DOI: 10.21105/joss.03021 Accessed: Mar. 31, 2026 (cit. on p. 19).

[38] Red Hat. "Ansible Collaborative," Accessed: Jan. 14, 2026. [Online]. Available: https://www.redhat.com/en/ansible-collaborative (cit. on p. 19).

[39] M. Kaiser. "SEG-UNIBE/artio-orchestration: Private repo for CI/CD orchestrations.," GitHub, Accessed: Jan. 14, 2026. [Online]. Available: https://github.com/SEG-UNIBE/artio-orchestration (cit. on pp. 19, 57).

[40] Cloudflare Inc. "Connect, protect, and build everywhere," Accessed: Jan. 14, 2026. [Online]. Available: https://www.cloudflare.com/en-gb/ (cit. on pp. 20, 49).

[41] J. Appelbaum and A. Muffett, *The ".onion" special-use domain name*, RFC 7686, RFC Editor, Oct. 2015. DOI: 10.17487/RFC7686 (cit. on p. 24).

[42] R. Moskowitz, D. Karrenberg, Y. Rekhter, E. Lear, and G. J. de Groot, *Address allocation for private internets*, RFC 1918, RFC Editor, Feb. 1996. DOI: 10.17487/RFC1918 (cit. on p. 24).

[43] T. Berners-Lee, R. T. Fielding, and L. M. Masinter, *Uniform resource identifier (URI): Generic syntax*, RFC 3986, RFC Editor, Jan. 2005. DOI: 10.17487/RFC3986 (cit. on p. 24).

[44] M. Cotton and L. Vegoda, *Special use ipv4 addresses*, RFC 5735, RFC Editor, Jan. 2010. DOI: 10.17487/RFC5735 (cit. on pp. 24, 53).

[45] J. Weil, V. Kuarsingh, C. Donley, C. Liljenstolpe, and M. Azinger, *IANA-reserved ipv4 prefix for shared address space*, RFC 6598, RFC Editor, Apr. 2012. DOI: 10.17487/RFC6598 (cit. on p. 24).

[46] D. Hoyte, *Hoytech/strfry*, Mar. 1, 2026. Accessed: Mar. 1, 2026. [Online]. Available: https://github.com/hoytech/strfry (cit. on pp. 25, 46).

[47] G. Heartsfield, *Scsibug/nostr-rs-relay*, Feb. 26, 2026. Accessed: Mar. 1, 2026. [Online]. Available: https://github.com/scsibug/nostr-rs-relay (cit. on p. 25).

[48] B. Deen, *Barrydeen/haven*, Feb. 28, 2026. Accessed: Mar. 1, 2026. [Online]. Available: https://github.com/barrydeen/haven (cit. on p. 25).

[49] J. Rubin, K. Czarnecki, and M. Chechik, "Managing cloned variants: A framework and experience," in *Proceedings of the 17th International Software Product Line Conference*, Tokyo Japan: ACM, Aug. 26, 2013, pp. 101–110, ISBN: 978-1-4503-1968-3. DOI: 10.1145/2491627.2491644 Accessed: May 2, 2026 (cit. on p. 27).

[50] M. Kaiser. "Development by ooemperor · Pull Request #16 · SEG-UNIBE/artio-relay," GitHub, Accessed: Mar. 17, 2026. [Online]. Available: https://github.com/SEG-UNIBE/artio-relay/pull/16 (cit. on p. 37).

[51] nostr-protocol. "Nips/46.md at master · nostr-protocol/nips," GitHub, Accessed: Mar. 25, 2026. [Online]. Available: https://github.com/nostr-protocol/nips/blob/master/46.md (cit. on p. 42).

[52] S. Cheshire and M. Krochmal, "Multicast DNS," RFC Editor, RFC6762, Feb. 2013, RFC6762. DOI: 10.17487/rfc6762 Accessed: Mar. 18, 2026 (cit. on pp. 43, 55).

[53] "Library of Alexandria," Alexandria, Accessed: Mar. 31, 2026. [Online]. Available: https://gitcitadel.eu/ (cit. on pp. 48, 54).

[54] Google. "IPv6 – Google," Google IPv6, Accessed: Jan. 14, 2026. [Online]. Available: https://www.google.com/intl/en/ipv6/statistics.html#tab=ipv6-adoption (cit. on p. 51).

[55] A. Vakali and G. Pallis, "Content delivery networks: Status and trends," *IEEE Internet Computing*, vol. 7, no. 6, pp. 68–74, Nov. 2003. DOI: 10.1109/MIC.2003.1250586 (cit. on p. 52).

[56] "Theforest.nostr1.com/," Accessed: Mar. 31, 2026. [Online]. Available: https://theforest.nostr1.com/ (cit. on p. 54).

[57] "Apache Kafka," Accessed: Mar. 31, 2026. [Online]. Available: https://kafka.apache.org/ (cit. on p. 54).

[58] "MQTT - The Standard for IoT Messaging," Accessed: Mar. 31, 2026. [Online]. Available: https://mqtt.org/ (cit. on p. 54).

[59] HAProxy Technologies, LLC. "HAProxy Technologies — Powering the World's Busiest Applications," HAProxy Technologies, Accessed: Jan. 14, 2026. [Online]. Available: https://www.haproxy.com/ (cit. on p. 57).

[60] "Nostr.watch/," nostr.watch, Accessed: May 2, 2026. [Online]. Available: https://nostr.watch/ (cit. on p. 61).

A
Appendix

A.1 Full UDM Data Flow Diagram

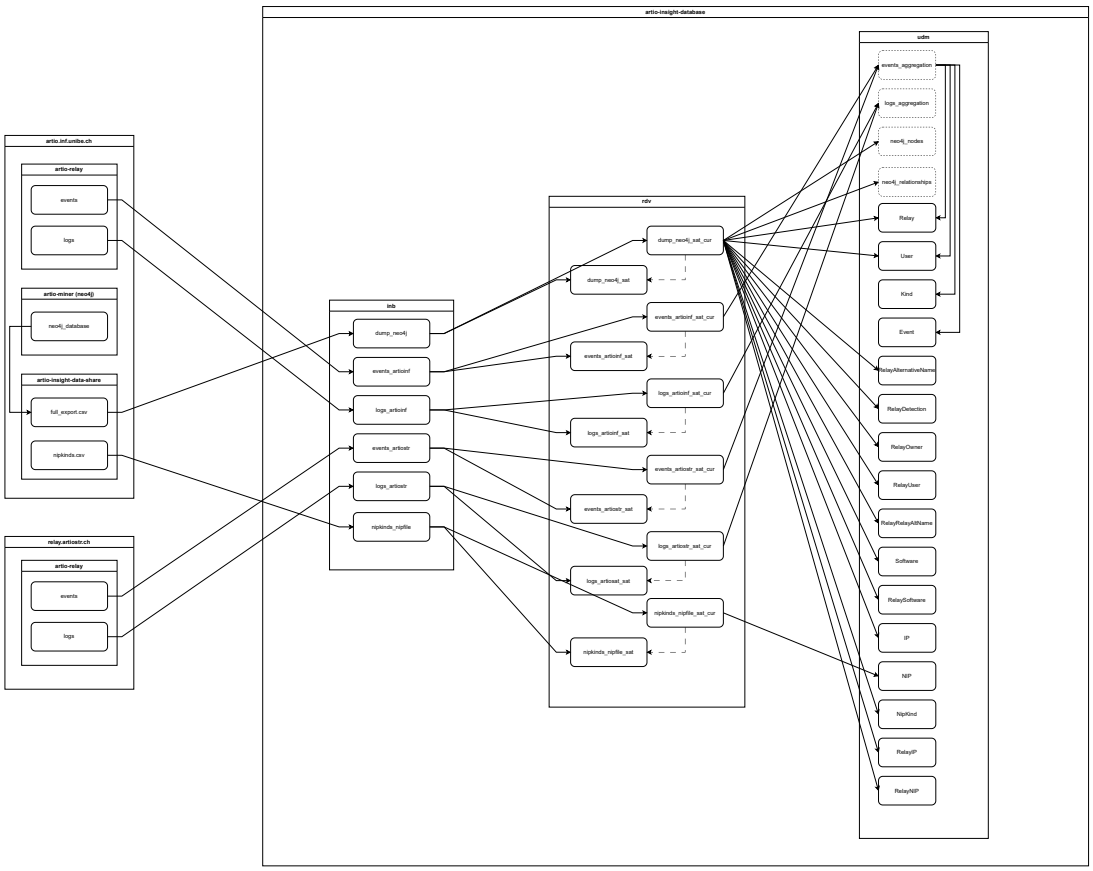


Figure A.1: Full Artio Data Flow Overview

A.2 SQL Code Examples For UDM Load Procedures

```
/*
============================================
Author:           Michael Kaiser
Create date:      2026-01-21
Description:      Procedure creation for running all udm procedures in the database
Modification:
============================================
*/

CREATE OR REPLACE PROCEDURE udm.run_all_loads(p_force boolean DEFAULT FALSE)
    LANGUAGE plpgsql
AS
$proc$
DECLARE
    r    RECORD;
    call_sql     TEXT;
    msg  TEXT;
    called_count INT := 0;
    failed_count INT := 0;
BEGIN
    -- Verify schema exists
    IF NOT EXISTS (SELECT 1 FROM pg_namespace WHERE nspname = 'udm') THEN
RAISE EXCEPTION 'Schema udm does not exist';
    END IF;

    FOR r IN
SELECT routine_name
FROM information_schema.routines
WHERE routine_schema = 'udm'
  AND routine_type = 'PROCEDURE'
  AND routine_name <> 'run_all_loads'
ORDER BY routine_name
LOOP
    -- Build CALL SQL, pass boolean literal
    call_sql := FORMAT('CALL udm.%I(%s)', r.routine_name, CASE WHEN p_force THEN 'true' ELSE '
     false' END);
    RAISE INFO 'Calling %', call_sql;
    BEGIN
EXECUTE call_sql;
called_count := called_count + 1;
RAISE INFO $$Succeeded: %$$, r.routine_name;
    EXCEPTION
WHEN OTHERS THEN
    failed_count := failed_count + 1;
    msg := FORMAT('Procedure %s failed: %s', r.routine_name, SQLERRM);
    RAISE INFO $$%$$, msg;
    END;
END LOOP;

    IF p_force THEN
GRANT SELECT ON ALL TABLES IN SCHEMA udm TO udm_reader;
GRANT USAGE ON SCHEMA udm TO udm_reader;

GRANT ALL ON ALL TABLES IN SCHEMA udm TO udm_admin;
GRANT USAGE ON SCHEMA udm TO udm_admin;
    END IF;

    msg := FORMAT('udm.run_all_loads finished: called=%s called, failed=%s failed', called_count,
     failed_count);
    RAISE INFO $$%$$, msg;
END;
$proc$;
```

Listing A.1: Code For Creating The Procedure For Loading All UDM Tables

```
/*
=============================================
Author:           Michael Kaiser
Create date:      2025-10-19
Description:      Table for the kinds of events
Modification:
2025-11-25  mkaiser     Changing the script to physical table
2026-01-12  mkaiser     Refactoring to procedure instead of loading script
=============================================
*/

CREATE OR REPLACE PROCEDURE udm.kind_load(p_force boolean DEFAULT FALSE)
    LANGUAGE plpgsql
AS
$proc$
BEGIN
    IF NOT EXISTS (SELECT 1 FROM pg_namespace WHERE nspname = 'udm') THEN
RAISE EXCEPTION 'Schema udm does not exist';
    END IF;

    IF p_force THEN
IF EXISTS (SELECT 1 FROM information_schema.tables WHERE table_schema = 'udm' AND table_name = '
     kind') THEN
    EXECUTE 'DROP TABLE udm.kind';
END IF;
    END IF;
    IF NOT EXISTS (SELECT 1 FROM information_schema.tables WHERE table_schema = 'udm' AND
     table_name = 'kind') THEN
EXECUTE $$CREATE TABLE udm.kind AS
      SELECT record_src,
     kind,
     COUNT(DISTINCT pubkey) AS user_count,
     COUNT(event_id)AS event_count
      FROM udm.events_aggregation
      GROUP BY kind, record_src$$;
EXECUTE 'CREATE INDEX IF NOT EXISTS idx_udm_kind_kind ON udm.kind (kind)';
    ELSE
BEGIN
    EXECUTE 'TRUNCATE TABLE udm.kind';
    EXECUTE $$INSERT INTO udm.kind (record_src, kind, user_count, event_count)
  SELECT record_src,
 kind,
 COUNT(DISTINCT pubkey) AS user_count,
 COUNT(event_id)AS event_count
  FROM udm.events_aggregation
  GROUP BY kind, record_src$$;
EXCEPTION
    WHEN OTHERS THEN
RAISE NOTICE 'Error while refreshing udm.kind: %', SQLERRM;
RAISE;
END;
    END IF;
END;
$proc$;
```

Listing A.2: Example Of A Load Procedure For The UDM.Kind Table

A.3 Mapping Data For The Event Kind To The Correct NIP.

Table A.1: Event Kind to NIP mapping based on [18]

kind	name	description	defined_in	nip_name
0	User Metadata	User Metadata	NIP-01	01
1	Short Text Note	Short Text Note	NIP-10	10
2	Recommend Relay	Recommend Relay	NIP-01	01 (deprecated)
3	Follows	Follows	NIP-02	02
4	Encrypted Direct Messages	Encrypted Direct Messages	NIP-04	04
5	Event Deletion Request	Event Deletion Request	NIP-09	09
6	Repost	Repost	NIP-18	18
7	Reaction	Reaction	NIP-25	25
8	Badge Award	Badge Award	NIP-58	58
9	Chat Message	Chat Message	C7	C7
10	Group Chat Threaded Reply	Group Chat Threaded Reply	NIP-29	29 (deprecated)
11	Thread	Thread	7D	7D
12	Group Thread Reply	Group Thread Reply	NIP-29	29 (deprecated)
13	Seal	Seal	NIP-59	59
14	Direct Message	Direct Message	NIP-17	17
15	File Message	File Message	NIP-17	17
16	Generic Repost	Generic Repost	NIP-18	18
17	Reaction to a website	Reaction to a website	NIP-25	25
20	Picture	Picture	NIP-68	68
21	Video Event	Video Event	NIP-71	71
22	Short-form Portrait Video Event	Short-form Portrait Video Event	NIP-71	71
24	Public Message	Public Message	A4	A4
30	internal reference	internal reference	NKBIP-03	NKBIP-03
31	external web reference	external web reference	NKBIP-03	NKBIP-03

Continued on next page

Table A.1: Event Kind to NIP mapping based on [18] (Continued)

kind	name	description	defined_in	nip_name
32	hardcopy reference	hardcopy reference	NKBIP-03	NKBIP-03
33	prompt reference	prompt reference	NKBIP-03	NKBIP-03
40	Channel Creation	Channel Creation	NIP-28	28
41	Channel Metadata	Channel Metadata	NIP-28	28
42	Channel Message	Channel Message	NIP-28	28
43	Channel Hide Message	Channel Hide Message	NIP-28	28
44	Channel Mute User	Channel Mute User	NIP-28	28
62	Request to Vanish	Request to Vanish	NIP-62	62
64	Chess (PGN)	Chess (PGN)	NIP-64	64
443	KeyPackage	KeyPackage	Marmot	Marmot
444	Welcome Message	Welcome Message	Marmot	Marmot
445	Group Event	Group Event	Marmot	Marmot
818	Merge Requests	Merge Requests	NIP-54	54
1018	Poll Response	Poll Response	NIP-88	88
1021	Bid	Bid	NIP-15	15
1022	Bid confirmation	Bid confirmation	NIP-15	15
1040	OpenTimestamps	OpenTimestamps	NIP-3	3
1059	Gift Wrap	Gift Wrap	NIP-59	59
1063	File Metadata	File Metadata	NIP-94	94
1068	Poll	Poll	NIP-88	88
1111	Comment	Comment	NIP-22	22
1222	Voice Message	Voice Message	A0	A0
1244	Voice Message Comment	Voice Message Comment	A0	A0
1311	Live Chat Message	Live Chat Message	NIP-53	53
1337	Code Snippet	Code Snippet	C0	C0
1617	Patches	Patches	NIP-34	34
1618	Pull Requests	Pull Requests	NIP-34	34
1619	Pull Request Updates	Pull Request Updates	NIP-34	34

Continued on next page

Table A.1: Event Kind to NIP mapping based on [18] (Continued)

kind	name	description	defined_in	nip_name
1621	Issues	Issues	NIP-34	34
1622	Git Replies (deprecated)	Git Replies (deprecated)	NIP-34	34
1630	Status	Status	NIP-34	34
1631	Status	Status	NIP-34	34
1632	Status	Status	NIP-34	34
1633	Status	Status	NIP-34	34
1971	Problem Tracker	Problem Tracker	nostrocket	nostrocket
1984	Reporting	Reporting	NIP-56	56
1985	Label	Label	NIP-32	32
1986	Relay reviews	Relay reviews		
1987	AI Embeddings / Vector lists	AI Embeddings / Vector lists	NKBIP-02	NKBIP-02
2003	Torrent	Torrent	NIP-35	35
2004	Torrent Comment	Torrent Comment	NIP-35	35
2022	Coinjoin Pool	Coinjoin Pool	joinstr	joinstr
4550	Community Post Approval	Community Post Approval	NIP-72	72
7000	Job Feedback	Job Feedback	NIP-90	90
7374	Reserved Cashu Wallet Tokens	Reserved Cashu Wallet Tokens	NIP-60	60
7375	Cashu Wallet Tokens	Cashu Wallet Tokens	NIP-60	60
7376	Cashu Wallet History	Cashu Wallet History	NIP-60	60
7516	Geocache log	Geocache log	geocaching	geocaching
7517	Geocache proof of find	Geocache proof of find	geocaching	geocaching
8000	Add User	Add User	NIP-43	43
8001	Remove User	Remove User	NIP-43	43
9000	Group Control Events	Group Control Events	NIP-29	29
9001	Group Control Events	Group Control Events	NIP-29	29

Continued on next page

Table A.1: Event Kind to NIP mapping based on [18] (Continued)

kind	name	description	defined_in	nip_name
9002	Group Control Events	Group Control Events	NIP-29	29
9003	Group Control Events	Group Control Events	NIP-29	29
9004	Group Control Events	Group Control Events	NIP-29	29
9005	Group Control Events	Group Control Events	NIP-29	29
9006	Group Control Events	Group Control Events	NIP-29	29
9007	Group Control Events	Group Control Events	NIP-29	29
9008	Group Control Events	Group Control Events	NIP-29	29
9009	Group Control Events	Group Control Events	NIP-29	29
9010	Group Control Events	Group Control Events	NIP-29	29
9011	Group Control Events	Group Control Events	NIP-29	29
9012	Group Control Events	Group Control Events	NIP-29	29
9013	Group Control Events	Group Control Events	NIP-29	29
9014	Group Control Events	Group Control Events	NIP-29	29
9015	Group Control Events	Group Control Events	NIP-29	29
9016	Group Control Events	Group Control Events	NIP-29	29
9017	Group Control Events	Group Control Events	NIP-29	29
9018	Group Control Events	Group Control Events	NIP-29	29
9019	Group Control Events	Group Control Events	NIP-29	29
9020	Group Control Events	Group Control Events	NIP-29	29

Continued on next page

Table A.1: Event Kind to NIP mapping based on [18] (Continued)

kind	name	description	defined_in	nip_name
9021	Group Control Events	Group Control Events	NIP-29	29
9022	Group Control Events	Group Control Events	NIP-29	29
9023	Group Control Events	Group Control Events	NIP-29	29
9024	Group Control Events	Group Control Events	NIP-29	29
9025	Group Control Events	Group Control Events	NIP-29	29
9026	Group Control Events	Group Control Events	NIP-29	29
9027	Group Control Events	Group Control Events	NIP-29	29
9028	Group Control Events	Group Control Events	NIP-29	29
9029	Group Control Events	Group Control Events	NIP-29	29
9030	Group Control Events	Group Control Events	NIP-29	29
9041	Zap Goal	Zap Goal	NIP-75	75
9321	Nutzap	Nutzap	NIP-61	61
9467	Tidal login	Tidal login	Tidal-nostr	Tidal-nostr
9734	Zap Request	Zap Request	NIP-57	57
9735	Zap	Zap	NIP-57	57
9802	Highlights	Highlights	NIP-84	84
10000	Mute list	Mute list	NIP-51	51
10001	Pin list	Pin list	NIP-51	51
10002	Relay List Metadata	Relay List Metadata	NIP-65	65
10003	Bookmark list	Bookmark list	NIP-51	51
10004	Communities list	Communities list	NIP-51	51
10005	Public chats list	Public chats list	NIP-51	51
10006	Blocked relays list	Blocked relays list	NIP-51	51
10007	Search relays list	Search relays list	NIP-51	51

Continued on next page

Table A.1: Event Kind to NIP mapping based on [18] (Continued)

kind	name	description	defined_in	nip_name
10009	User groups	User groups	NIP-51	51
10011	External Identities	External Identities	NIP-39	39
10012	Favorite relays list	Favorite relays list	NIP-51	51
10013	Private event relay list	Private event relay list	NIP-37	37
10015	Interests list	Interests list	NIP-51	51
10019	Nutzap Mint Recommendation	Nutzap Mint Recommendation	NIP-61	61
10020	Media follows	Media follows	NIP-51	51
10030	User emoji list	User emoji list	NIP-51	51
10050	Relay list to receive DMs	Relay list to receive DMs	51	51
10051	KeyPackage Relays List	KeyPackage Relays List	Marmot	Marmot
10063	User server list	User server list	Blossom	Blossom
10096	File storage server list	File storage server list	NIP-96	96 (deprecated)
10166	Relay Monitor Announcement	Relay Monitor Announcement	NIP-66	66
10312	Room Presence	Room Presence	NIP-53	53
10377	Proxy Announcement	Proxy Announcement	Nostr Epoxy	Nostr Epoxy
11111	Transport Method Announcement	Transport Method Announcement	Nostr Epoxy	Nostr Epoxy
13194	Wallet Info	Wallet Info	NIP-47	47
13534	Membership Lists	Membership Lists	NIP-43	43
14388	User Sound Effect Lists	User Sound Effect Lists	Corny Chat	Corny Chat
17375	Cashu Wallet Event	Cashu Wallet Event	NIP-60	60
21000	Lightning Pub RPC	Lightning Pub RPC	Lightning.Pub	Lightning.Pub
22242	Client Authentication	Client Authentication	NIP-42	42
23194	Wallet Request	Wallet Request	NIP-47	47

Continued on next page

Table A.1: Event Kind to NIP mapping based on [18] (Continued)

kind	name	description	defined_in	nip_name
23195	Wallet Response	Wallet Response	NIP-47	47
24133	Nostr Connect	Nostr Connect	NIP-46	46
24242	Blobs stored on mediaservers	Blobs stored on mediaservers	Blossom	Blossom
27235	HTTP Auth	HTTP Auth	NIP-98	98
28934	Join Request	Join Request	NIP-43	43
28935	Invite Request	Invite Request	NIP-43	43
28936	Leave Request	Leave Request	NIP-43	43
30000	Follow sets	Follow sets	NIP-51	51
30001	Generic lists	Generic lists	NIP-51	51 (deprecated)
30002	Relay sets	Relay sets	NIP-51	51
30003	Bookmark sets	Bookmark sets	NIP-51	51
30004	Curation sets	Curation sets	NIP-51	51
30005	Video sets	Video sets	NIP-51	51
30006	Picture sets	Picture sets	NIP-51	51
30007	Kind mute sets	Kind mute sets	NIP-51	51
30008	Profile Badges	Profile Badges	NIP-58	58
30009	Badge Definition	Badge Definition	NIP-58	58
30015	Interest sets	Interest sets	NIP-51	51
30017	Create or update a stall	Create or update a stall	NIP-15	15
30018	Create or update a product	Create or update a product	NIP-15	15
30019	Marketplace UI/UX	Marketplace UI/UX	NIP-15	15
30020	Product sold as an auction	Product sold as an auction	NIP-15	15
30023	Long-form Content	Long-form Content	NIP-23	23
30024	Draft Long-form Content	Draft Long-form Content	NIP-23	23
30030	Emoji sets	Emoji sets	NIP-51	51
30040	Curated Publication Index	Curated Publication Index	NKBIP-01	NKBIP-01

Continued on next page

Table A.1: Event Kind to NIP mapping based on [18] (Continued)

kind	name	description	defined_in	nip_name
30041	Curated Publication Content	Curated Publication Content	NKBIP-01	NKBIP-01
30063	Release artifact sets	Release artifact sets	NIP-51	51
30078	Application-specific Data	Application-specific Data	NIP-78	78
30166	Relay Discovery	Relay Discovery	NIP-66	66
30267	App curation sets	App curation sets	NIP-51	51
30311	Live Event	Live Event	NIP-53	53
30312	Interactive Room	Interactive Room	NIP-53	53
30313	Conference Event	Conference Event	NIP-53	53
30315	User Statuses	User Statuses	NIP-38	38
30382	User Trusted Assertion	User Trusted Assertion	NIP-85	85
30383	Event Trusted Assertion	Event Trusted Assertion	NIP-85	85
30384	Addressable Trusted Assertion	Addressable Trusted Assertion	NIP-85	85
30388	Slide Set	Slide Set	Corny Chat	Corny Chat
30402	Classified Listing	Classified Listing	NIP-99	99
30403	Draft Classified Listing	Draft Classified Listing	NIP-99	99
30617	Repository announcements	Repository announcements	NIP-34	34
30618	Repository state announcements	Repository state announcements	NIP-34	34
30818	Wiki article	Wiki article	NIP-54	54
30819	Redirects	Redirects	NIP-54	54
31234	Draft Event	Draft Event	NIP-37	37
31388	Link Set	Link Set	Corny Chat	Corny Chat
31890	Feed	Feed	NUD: Custom Feeds	NUD: Custom Feeds
31922	Date-Based Calendar Event	Date-Based Calendar Event	NIP-52	52
31923	Time-Based Calendar Event	Time-Based Calendar Event	NIP-52	52

Continued on next page

Table A.1: Event Kind to NIP mapping based on [18] (Continued)

kind	name	description	defined_in	nip_name
31924	Calendar	Calendar	NIP-52	52
31925	Calendar Event RSVP	Calendar Event RSVP	NIP-52	52
31989	Handler recommendation	Handler recommendation	NIP-89	89
31990	Handler information	Handler information	NIP-89	89
32267	Software Application	Software Application		
32388	User Room Favorites	User Room Favorites	Corny Chat	Corny Chat
33388	High Scores	High Scores	Corny Chat	Corny Chat
34235	Addressable Video Event	Addressable Video Event	NIP-71	71
34236	Addressable Short Video Event	Addressable Short Video Event	NIP-71	71
34388	Sound Effects	Sound Effects	Corny Chat	Corny Chat
34550	Community Definition	Community Definition	NIP-72	72
38172	Cashu Mint Announcement	Cashu Mint Announcement	NIP-87	87
38173	Fedimint Announcement	Fedimint Announcement	NIP-87	87
37516	Geocache listing	Geocache listing	geocaching	geocaching
38383	Peer-to-peer Order events	Peer-to-peer Order events	NIP-69	69
39000	Group metadata events	Group metadata events	NIP-69	69
39001	Group metadata events	Group metadata events	NIP-69	69
39002	Group metadata events	Group metadata events	NIP-69	69
39003	Group metadata events	Group metadata events	NIP-69	69
39004	Group metadata events	Group metadata events	NIP-69	69

Continued on next page

Table A.1: Event Kind to NIP mapping based on [18] (Continued)

kind	name	description	defined_in	nip_name
39005	Group metadata events	Group metadata events	NIP-69	69
39006	Group metadata events	Group metadata events	NIP-69	69
39007	Group metadata events	Group metadata events	NIP-69	69
39008	Group metadata events	Group metadata events	NIP-69	69
39009	Group metadata events	Group metadata events	NIP-69	69
39089	Starter packs	Starter packs		51
39092	Media starter packs	Media starter packs		51
39701	Web bookmarks	Web bookmarks		B0

Erklärung

gemäss Art. 30 RSL Phil.-nat.18

Name/Vorname: Kaiser Michael

Matrikelnummer: 17-115-718

Studiengang: Informatik

Bachelor ☐ Master ☑ Dissertation ☐

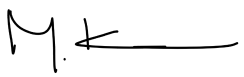
Titel der Arbeit: Artio: Large Scale Investigation of Software Variability in Nostr

LeiterIn der Arbeit: Prof. Dr. Timo Kehrer
Roman Bögli

Ich erkläre hiermit, dass ich diese Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen benutzt habe. Alle Stellen, die wörtlich oder sinngemäss aus Quellen entnommen wurden, habe ich als solche gekennzeichnet. Mir ist bekannt, dass andernfalls der Senat gemäss Artikel 36 Absatz 1 Buchstabe r des Gesetzes vom 5. September 1996 über die Universität zum Entzug des auf Grund dieser Arbeit verliehenen Titels berechtigt ist.
Für die Zwecke der Begutachtung und der Überprüfung der Einhaltung der Selbständigkeitserklärung bzw. der Reglemente betreffend Plagiate erteile ich der Universität Bern das Recht, die dazu erforderlichen Personendaten zu bearbeiten und Nutzungshandlungen vorzunehmen, insbesondere die schriftliche Arbeit zu vervielfältigen und dauerhaft in einer Datenbank zu speichern sowie diese zur Überprüfung von Arbeiten Dritter zu verwenden oder hierzu zur Verfügung zu stellen.

Herzogenbuchsee, 20. Mai 2026

Ort/Datum



Unterschrift