Community-Driven Variability: Characterizing a new Software Variability Paradigm

Roman Bögli[1*], Alexander Boll[1], Alexander Schultheiß[1], Timo Kehrer[1]

[1]Institute of Computer Science, University of Bern, Bern, Switzerland.

*Corresponding author(s). E-mail(s): roman.boegli@unibe.ch;
Contributing authors: alexander.boll@unibe.ch;
alexanderschultheiss@pm.me; timo.kehrer@unibe.ch;

Abstract

Both software engineering researchers and practitioners have increasingly shifted their focus from single software systems to software families, reflecting the need for software industrialization through systematic reuse of implementation artifacts. Interestingly, several vibrant ecosystems produce software families in a radically different way than classical variability-intensive systems, notably software product lines (SPLs). The Bitcoin community, for instance, evolves its ecosystem through crowdsourced improvement proposals being continuously shaped and autonomously implemented by independent actors. While this novel paradigm of Community-Driven Variability (CDV) has proven effective for driving flourishing technologies like Bitcoin and others, it also comes with unique challenges calling for novel solutions. In this paper, we define the key characteristics of ecosystems exposing CDV and derive a taxonomy that hierarchically decomposes each characteristic into constituting sub-characteristics. Building on the taxonomy, we conduct a systematic analysis of 14 software ecosystems to evaluate the presence and nature of CDV. We highlight the novel problems they face, such as the lack of ecosystem overview, difficulties in impact assessment, misalignment between proposals and implementations, and interoperability breakdowns – challenges that transcend classical variability management. Based on the problem analysis, we outline our research vision to tackle these challenges, including a sketch of concrete starting points for technical solutions. While classical SPLs and CDV ecosystems differ drastically, we believe that feature-oriented modeling and analysis offers promising concepts for addressing CDV challenges without enforcing product-line processes. Conversely, the unique demands of CDV can inspire advances in variability research with impact beyond its original domains.

Keywords: software families, software variability, improvement proposals, implementation derivatives, interoperability, evolution

1 Introduction

Since Parnas' seminal work on program families in the 1970s [1], both software engineering researchers and practitioners have increasingly shifted their focus from developing single software systems to managing families of software variants sharing common functionality [2]. The variants in a software family share common functionality, motivated by the need to accommodate varying requirements across different markets, customer needs, or operational environments [2]. Today, the impact of Parnas' work is seen everywhere in modern software engineering, ranging from agile modular design to preplanning-intensive platforms for manufacturing highly customized software variants. The most systematic class of approaches for developing such variability-intensive systems is summarized under the umbrella term of *software product-line* (SPL) engineering [2, 3], which relies on an explicit model of variability in terms of features realized based on an integrated software platform [4, 5]. More specifically, an SPL is implemented by mapping features onto implementation artifacts and choosing a variation mechanism which specifies how to generate individual products [3, 5, 6]. Success stories of SPL adoption have been reported for various domains, notably for embedded control software [3] in automotive [7], aerospace [8], railway [9], and telecommunications [10], as well as and for systems software such as the Linux kernel [11]. Recent literature also discusses more liberal approaches to managing software families, spanning a continuum that ranges from managing ad-hoc clone-and-own [12–15] over flexible product-line adoption [16–21] to feature toggling [22, 23] in distributed open-source communities. Although they deviate from rigorous product-line engineering, all these approaches deal with software variability in one way or another.

While today's software families use countless different approaches to manage software variability, they share two common key characteristics. First, the main motivation for dealing with software variability is due to economic reasons. In essence, it involves adopting the principles of industrialization and mass customization from other engineering disciplines, with the goal of shortening time-to-market and reducing development and maintenance costs [3]. Second, managing software variability revolves around the fundamental principle of software reuse, albeit at varying levels of systematic organization and planning [24]. At its core, however, it is software reuse that avoids the redundant development and maintenance of software artifacts implementing common functionality of the members of a software family.

Interestingly, several vibrant ecosystems produce software families in a radically different way than classical variability-intensive systems. They are driven by factors other than software industrialization and mass customization, and exhibit variability that is not focused on reusing implementation artifacts nor centrally managed. Instead, they focus on achieving interoperability within the software family through the ecosystem community's continuous effort to shape an open set of specification documents, referred to as *improvement proposals* (IPs). Based on this set of IPs, developer groups within the community independently derive their own variants by selecting and implementing a desired subset of IPs. This independent derivation fosters a broad range of software variability across multiple dimensions.

```
  BIP:              <BIP number, or "?" before being assigned>
* Layer:            <Consensus (soft fork) | Consensus (hard fork) | Peer Services |
                     API/RPC | Applications>
  Title:            <BIP title; maximum 44 characters>
  Author:           <list of authors' real names and email addrs>
* Discussions-To:   <email address>
* Comments-Summary: <summary tone>
  Comments-URI:     <links to wiki page for comments>
  Status:           <Draft | Active | Proposed | Deferred | Rejected | Withdrawn |
                     Final | Replaced | Obsolete>
  Type:             <Standards Track | Informational | Process>
  Created:          <date created on, in ISO 8601 (yyyy-mm-dd) format>
  License:          <abbreviation for approved license(s)>
* License-Code:     <abbreviation for code under different approved license(s)>
* Post-History:     <dates of postings to bitcoin mailing list, or link to thread in
                     mailing list archive>
* Requires:         <BIP number(s)>
* Replaces:         <BIP number>
* Superseded-By:    <BIP number>
```

Figure 1: BIP preamble structure from BIP2 [25].

As an example for such an ecosystem, consider Bitcoin [26] with its various application types (e.g., nodes, wallets, block explorers, side-chains) and actors (e.g., developers, users, analysts). The concepts that define Bitcoin, along with any potential features introduced to the ecosystem, are shaped by *Bitcoin Improvement Proposals* (BIPs) [27], a decentralized collection of open-source specification documents written by independent actors sharing mutual interests. The structure and process for proposing, approving, discarding, and managing BIPs is itself also specified in this manner, specifically in BIP2 [25]. An excerpt of this BIP specification is presented in Figure 1, showing the template preamble each BIP should follow. Developers independently choose and implement subsets of BIPs in their applications, yielding a constantly growing set of software variants to which we refer as *implementation derivatives*. These derivatives may address different use cases (e.g., nodes, wallets, exchanges, watchtowers, block explorers) even though they are derived from a common set of BIPs. Conceptually, the commonalities and differences among these derivatives can be partially described in terms of BIPs, but there is typically no reuse of development artifacts at the implementation level. In fact, the variable dimension even spans over to the technology stack used to implement the derivatives, making classical code reuse less applicable. Nevertheless, we see the ecosystem evolving with incredible dynamism, exposing multidimensional variability to which we refer as *Community-Driven Variability* (CDV).

Beyond Bitcoin, this novel form of CDV also appears in other ecosystems outside the digital money domain, each applying a slightly different interpretation of IPs. Examples include the InterPlanetary File System (IPFS) [28] with *InterPlanetary Improvement Proposals* (IPIPs) [29], The Onion Router (Tor) [30] with its *design proposals* (TorDPs) [31], and Nostr [32], a decentralized protocol for secure message

exchange via cryptography and distributed relays, which uses *Nostr Implementation Possibilities* (NIPs) [33]. Section 5 revisits these examples in detail, along with additional ecosystems from both within and outside the digital money domain.

The paradigm of continuously shaping a de-facto standard and its implementation derivatives has proven to be an effective method for evolving open-source technologies with significant dynamism and traction. Yet, these ecosystems not only encounter challenges similar to those of classical variability-intensive systems, but also entirely new ones. Without an explicit variability model, managing the consistent evolution of IPs becomes increasingly challenging and error-prone. BIP2, for example, has recently (Sep. 18, 2024) received a revision request [34] motivated by several *"pain points"*. This indicates the need to improve the governance of the decentralized proposal process, addressing growing challenges wrt. maintaining overview, transparency, and consensus within the current proposal framework. Furthermore, derivatives may expose impaired interoperability, which is usually not the case for classical software families where variants are meant to be standalone software products. For example, a Reddit post [35] raises awareness for incompatibility issues induced by *BIP32 HD Wallets*. This BIP proposed a way to deterministically derive a hierarchy of asymmetric key pairs from a single secret [36]. Follow-up discussions on Bitcoin Stack Exchange [37] and a wallet recovery site [38] underscore the issue's severity. In addition, various online resources exist for curating, comparing, and recommending wallet applications [39–42]. These handcrafted ad-hoc monitoring efforts underscore both the richness of existing variability and, more importantly, the need to manage it effectively. Yet, this need remains largely unaddressed and offers substantial potential for systematic, especially automated, solutions that would benefit the community.

While classical SPL domains and emerging technologies following the CDV paradigm appear miles apart, we recognize the value in exploring this novel paradigm and the possibility of adapting concepts from one paradigm to the other. Since the use of features as a central domain abstraction in SPLs aligns well with IPs in CDV, adapting feature-oriented modeling and analysis seems promising for tackling CDV-induced problems without necessitating the adoption of product-line development processes. Conversely, research on classical variability-intensive systems will gain new momentum through the unique problems posed by CDV, leading to advancements that will push the state-of-the-art and generate new insights that may ultimately influence other domains.

In this paper, we outline our research vision on entering the novel field of CDV, summarizing our contributions as follows:

- We introduce the concept of CDV and describe this emerging paradigm using our motivating example of Bitcoin in Section 3.
- Based on these observations, we derive the generalized defining characteristics in Section 4, constituting a taxonomy to systematically evaluate whether a given ecosystem is subject to CDV or not.
- We employ this taxonomy on a set of case studies in Section 5 to scrutinize this CDV-detecting methodology and classify other CDV exhibiting ecosystem.

- Having defined and extensively studied its constituting characteristics, we examine key problems faced in ecosystems that exhibit CDV (Section 6).
- We derive our research vision and concrete research goals to address the key problems and outline our next steps to accomplish them (Section 7).

Succeeding our initial short paper on CDV [43], this paper significantly extends the scope and depth of our analysis of CDV ecosystems and their characteristics. In particular, we extend our previous work in the following ways:

- We enrich our initial case study of Bitcoin as a prominent example of CDV with several additional examples from online resources, providing a more nuanced view of CDV dynamics in practice.
- We provide foundational background on variability-intensive software systems in a dedicated section, thereby improving the self-contained nature of this work.
- Rather than generalizing CDV characteristics through informal descriptions, we now present a taxonomy that hierarchically decomposes each characteristic into defining sub-characteristics.
- Building on this taxonomy, we perform a thorough analysis of 14 software ecosystems, evaluating the presence and nature of CDV characteristics and sub-characteristics. This replaces our previous high-level comparison between classical software families and CDV-based ecosystems.
- We broaden our analysis beyond clearly differentiated cases (e.g., Bitcoin vs. SPL) by including ecosystems such as the Python programming language, where the distinction between CDV and classical variability is more subtle.
- Finally, we refine our research vision by concretizing promising starting points for future technical solutions aimed at advancing CDV in practice, with a particular focus on systematic and automated approaches.

2 Background, Motivation and Scope

Software variability can be understood both as a *phenomenon that arises naturally* and as a *capability intentionally designed* into software systems. The former, *descriptive* or *observational dimension*, reflects the insight that software systems inevitably change and diversify over time; an insight that dates back to the early days of software engineering and the emergence of the discipline itself (Sec. 2.1). The latter, *prescriptive* or *constructive* dimension, considers variability as a design concern or software quality; an ambition that has shaped software engineering practice for decades and remains an active software engineering research area today (Sec. 2.2). While a comprehensive survey is beyond the scope of this section, we recall a set of fundamental definitions representative of the two dimensions to underpin our subsequent discussion, and to motivate and position our work within the existing literature (Sec. 2.3).

2.1 Software Variability as a Natural Phenomenon

In his foreword of the edited book on software engineering for variability-intensive systems by Mistrik et al. [44], Grundy characterizes software variability as the *"expectation that computing systems will vary throughout their lifecycle"*, where variability may manifest through adaptation to, e.g., diverse domains and users, deployment on heterogeneous platforms, or ongoing organizational and environmental change. This characterization reflects what we refer to as the natural phenomenon of software variability as an intrinsic characteristic of the lifecycle of any non-trivial software system. Looking back further in history, although the term was not used as explicitly as it is today, software variability as a natural phenomenon lies at the heart of several seminal works and turning points in software engineering. For example, variability as a phenomenon was a key factor that led to the software crisis of the 1960s, which revealed the growing difficulty of maintaining and adapting increasingly complex software systems [45]. The observational view also aligns with Lehman's *Laws of Software Evolution*, first formulated in the 1970s, which identify continuous change as a defining characteristic of any real-world software system [46]. Today, it is commonly agreed that software variability is the manifestation of evolutionary pressure, i.e., a system's necessity to adapt. Looking at more recent literature, this perspective is reflected and refined in, e.g., the taxonomy proposed by Ananieva et al. [47], who use the terms variability and evolution somewhat interchangeably, distinguishing between software *variability (or evolution) in time and in space*. This distinction links the temporal dynamics of system change with the structural diversity of co-existing variants. In sum, all of these works articulate software variability as a descriptive concept, rooted in the observation of how software systems evolve and diversify in response to internal and external change.

2.2 Software Variability as a Capability

In contrast to the observational view, the taxonomy proposed by Svahnberg et al. defines software variability as *"the ability of a software system or artefact to be efficiently extended, changed, customized, or configured for use in a particular context"* [48]. This definition highlights that the software engineering community has long

recognized the need to address the limitations of treating variability as an unmanaged, emergent phenomenon, as the uncontrolled proliferation of software variants through simple, ad-hoc clone-and-own practices has been shown to result in redundancy, inconsistencies, and increased maintenance costs [49]. Consequently, rather than viewing diversity as an unavoidable side effect of evolution, researchers and practitioners began to treat software variability as a design concern that should be made explicit and systematically controlled. In a broader view, such proactive treatment of software variability is one of the contributing factors to overall improved evolvability, i.e., the ability to adapt to change [50, 51]. Numerous developments exemplify this paradigm shift, ranging from organizational transformations such as the replacement of waterfall-like processes by agile methods, over technological advances and the emergence of novel architectural and programming paradigms supporting flexibility and reuse, to the rise of continuous practices tightly integrated with tools like version control and CI/CD pipelines.

The transition from an observational to a constructive view of variability is manifested in Parnas' seminal work on program families in the 1970s [1], which argued that systems should be developed as families of related programs rather than as isolated products. This line of thinking laid the conceptual foundation for software product-line engineering (SPLE) [2, 3], which today can be seen as the most systematic and rigorous approach to treating variability as a *first-class engineering capability*. Software product lines (SPLs) promote the use of features as an *"optional or incremental unit of functionality"* [52] or as a *"product characteristic that is used in distinguishing programs within a family of related programs"* [53]. These definitions emphasize that features are supposed to provide a suitable abstraction for managing variability both *in time* (capturing progress over time in terms of newly introduced functionality) and *in space* (describing the commonalities and differences across a family of related, co-existing variants).

SPLs make features explicit throughout the entire system and software lifecycle, from requirements and architecture to implementation and testing. Accordingly, an SPL relies on an explicit model of variability in terms of features that are realized through an integrated software platform [4, 5]. Thereby, abstract features are mapped to concrete implementation artefacts, and a suitable variation mechanism specifies how individual products can be automatically derived from shared artifacts. Within the paradigm of SPLs, Apel et al. define software variability as *"the ability to derive different products from a common set of artefacts"* [3]. Similar to the definition of Svahnberg et al. [48], this view reframes variability from a natural consequence of software evolution into a deliberate design goal, emphasizing software reuse and the automated nature of product generation in SPLs.

Today, the fundamental principles of SPLs are well understood. However, the research field remains highly active, particularly in areas such as product-line testing and analysis [54], as ensuring the quality of millions or even billions of potential product variants remains one of the most significant challenges in SPLE. Another ongoing research frontier concerns the evolution of entire product lines over time, thereby realizing the vision of supporting variability both in space and in time [55]. The ongoing relevance of these challenges is reflected in major research venues such

as SPLC[1], VaMoS[2], and ICSR[3], which will merge in 2026 to form the VARIABILITY conference[4], highlighting the continued importance of research on software variability.

2.3 Research Motivation and Scope

In this paper, we introduce a new form of software variability that we term *Community-Driven Variability* (CDV). Building on the definitions of variability introduced above, we adopt a developer-centric perspective to study this phenomenon. Thereby, our focus lies on the observational dimension, in which we describe CDV and its defining characteristics. We then discuss a set of challenges that reveal how this form of variability lies in the tension between descriptive and constructive understandings of variability. Finally, we outline our research vision, arguing that principles from SPLE may inspire constructive approaches to addressing these emerging challenges in the future.

By establishing this vision, we lay the groundwork for a broader research agenda motivated by three key factors. First, CDV represents an unexplored form of variability that is radically different from established paradigms within the domain of variability-intensive systems. Second, it introduces unique challenges that warrant deeper analysis and novel technical solutions. Third, the fact that CDV is adopted across diverse and widely used software ecosystems underscores its practical relevance and the significant impact that advancements in this field can achieve.

[1] International Systems and Software Product Line Conference: https://splc.net
[2] International Working Conference on Variability Modelling of Software-Intensive Systems: https://vamosconf.net
[3] International Conference on Systems and Software Reuse; last edition in 2025: https://conf.researchr.org/home/icsr-2025
[4] International Conference on Software and Systems Reuse, Product Lines, and Configuration; first edition in 2026: https://conf.researchr.org/home/variability-2026

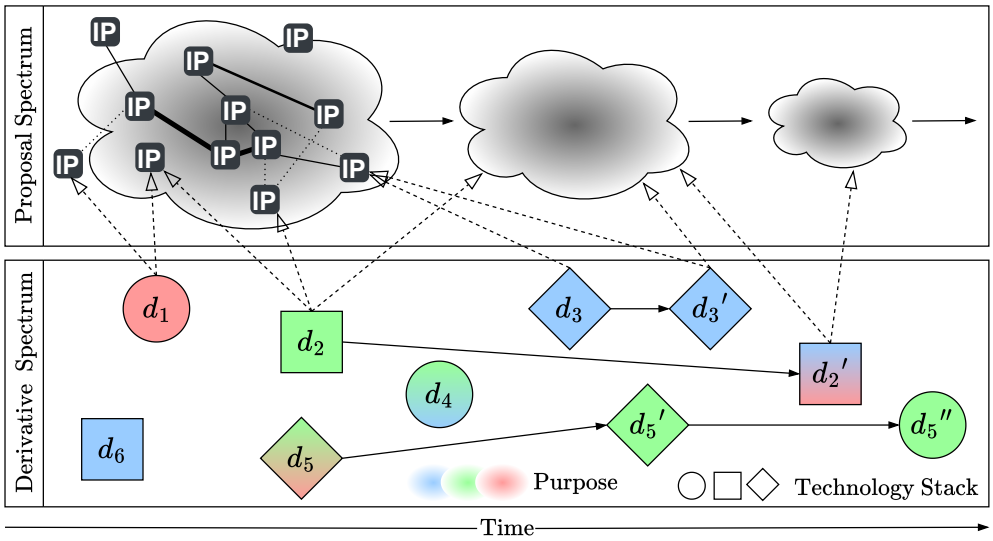


Figure 2: A schematic overview of the CDV landscape.

3 Showcasing CDV Characteristics in Bitcoin

Following on the descriptive perspective of software variability as a naturally occurring phenomenon outlined in Section 2.1, we ground our exploration of the emerging CDV paradigm in concrete observations drawn from practice. As a starting point, we begin with a focused examination of the Bitcoin ecosystem as our motivating example that initially inspired our work. Bitcoin, introduced in 2008 [26], is an electronic version of cash that operates without central authority. It relies on a peer-to-peer network and a public ledger, the blockchain, where data blocks are linked via cryptographic hashes and validated through distributed consensus [56]. Anyone may maintain a full copy of this ledger and issue transactions through one of the many available clients, commonly referred to as wallets.

Our examination of the Bitcoin ecosystem within the emerging CDV paradigm is grounded in online resources, supplemented by interviews with a Bitcoin derivative developer and an advanced end user. We illustrate a summary of our results in Figure 2. The proposal spectrum comprising the ecosystem's improvement proposals (IPs) is illustrated on top. The lower part illustrates the derivative spectrum comprising the ecosystem's applications, indicated as implementation derivatives $d_1 - d_6$ implementing varying sets of IPs. Both the proposal spectrum and the derivative spectrum evolve continuously, indicated by time progressing from left to right.

IPs are open-source specification documents written by independent actors sharing mutual interests. A substantial amount of IPs closely aligns with the traditional notion of features, with some even becoming synonymous with feature names. For instance, BIPs are reflected in the user interface (UI) of wallet applications such as *Sparrow* [57]. Figure 3 illustrates this by showing the wallet initialization wizard, where users can

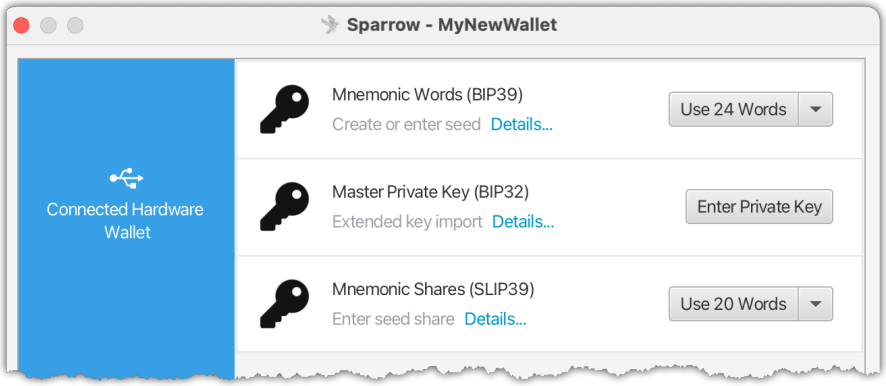


Figure 3: Sparrow [57] UI displaying BIP-based wallet creation features.

select from a list of features to configure a wallet according to their preferences. There, IP-based feature names such as *BIP32 HD Wallets* [36] and *BIP39 Mnemonic Seeds* [58], widely used in the Bitcoin community, are proactively mirrored in the interface itself.

Moreover, IPs have a dedicated status and may expose various kinds of interrelations (connection lines between IPs in Figure 2). BIP2 (cf. Figure 1), for instance, mentions status labels ranging from *draft* over *final* up to *replaced* or *obsolete*, and IP interrelations such as *requires*, *replaces*, or *superseded-by*. This indicates that IP statuses and interrelations are continuously reshaped, extended, overruled, or rejected. For example, BIP84 requires BIP173, while BIP173 has replaced BIP142 and itself is superseded by BIP350. While being similar in nature, other ecosystems may define a different set of IP statuses and may have other kinds of interrelations.

Applications constituting the derivative spectrum may be created at different points in time, each of them implementing an autonomously selected set of IPs (dashed arrows from d_n to IPs). While exposing variability in terms of conceptual features shaped by IPs, implementation derivatives can be built on various technology stacks and serve distinct or overlapping purposes. Figure 2 illustrates this range of technology and purpose using different shapes and gradient-colored backgrounds, respectively. We deliberately illustrate the IP set implemented by a derivative separately from its purpose, as the latter cannot always be inferred solely from the former. Some derivatives remain stable over time (d_1, d_6), while others may evolve. In the latter case, this evolution can unfold in various dimensions, e.g., the supported IP set ($d_3 \rightarrow d_3'$), a shift in the intended purpose (d_2'), or migrating to other technology stacks (d_5'').

From an organizational standpoint, derivatives possess full autonomy in composing their IP sets. However, to promote interoperability, the ecosystem's community typically maintains a shared understanding of the de-facto standard at any given time. We illustrate the evolving de-facto standard as a forward-moving IP cloud that may morph in shape over time. Core IPs serving as active building blocks for other dependent IPs are likely to be implemented more frequently by derivatives than others, and thus contribute to the perception of a de-facto standard (central part of an IP cloud in

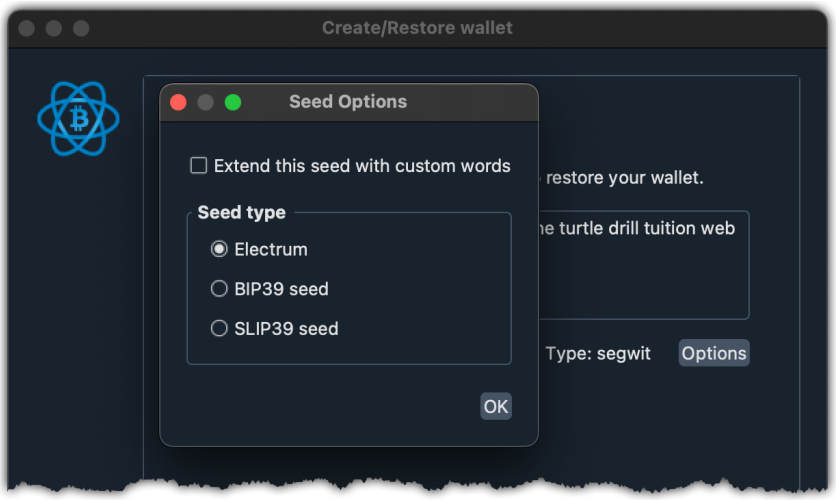


Figure 4: Electrum [59] UI displaying a variant of BIP-based wallet creation features, defaulting to its own implementation variant.

Figure 2). Outside this de-facto standard, there may be other IPs or informal proposals which are not (yet) officially approved but generally accepted by the community (outer part of an IP cloud). For instance, other renowned sources such as, e.g., the *Satoshi-Labs Improvement Proposals* (SLIPs) [60], augment the primary catalog of BIPs [27]. Likewise, IPs that are not yet finalized can still become part of de-facto standard if widely adopted. For example, the widespread implementation of the aforementioned mnemonic seeds according to BIP39, though officially holding "proposed" status until November 2024 (now classified as "final" [61]), has long been a standard feature among derivatives. Conversely, derivatives may counter established IPs using their own alternatives motivated by their own technological goals. The derivative *Electrum* [59], for instance, argues shortcomings in BIP39 and thus *"does not generate BIP39 seeds"* by default [62], advocating its own alternative as shown in Figure 4. Both Sparrow and Electrum additionally support a SLIP – specifically SLIP39 – which exemplifies the full freedom derivatives have in choosing which IPs to support, even if they originate from entirely different catalogs or sources, as is the case here with SatoshiLabs.

Together, these examples illustrate the considerable flexibility derivatives exercise in selecting which IPs to implement, even across catalogs. Moreover, these wallet creation methods reflect not only the diversity across implementation derivatives, but also the configurability of the derivatives themselves, a form of variability that closely resembles classical software variability on top of CDV.

4 Constituting Characterization of a Novel Paradigm

By generalizing from the insights in the previous section, we now define the constituting characteristics of ecosystems exhibiting CDV. Clarifying the core CDV traits lays the groundwork for recognizing whether CDV is present within a given ecosystem and to what extent. Further, establishing a taxonomy of characteristics from these traits provides the analytical foundation for developing automated approaches capable of supporting or amplifying CDV processes.

We derive the constituting characteristics through iterative open idea shaping. For the development of the characteristics, we use a broad and iterative three-step process that encompasses a variety of ecosystems, not limited to Bitcoin.

Step 1 – Discovery: In the first step, we examined ecosystems that implement or adopt peer-to-peer electronic coins or other decentralized systems with a similar ideological orientation. In addition, our consulted domain experts (e.g. Lightning developer) pointed out candidate ecosystems to discover. Lastly, we also discovered other ecosystems that follow a comparable IP-driven development culture, such as Tor with its Design Proposals (TorDPs) or Python with its Enhancement Proposals (PEPs).

Step 2 – Internal Validation: In the second step, we identified common and diverging characteristics among the set of discovered ecosystems in order to recognize defining observations that informed our shared understanding of CDV. The emerging taxonomy fragments were thereby constantly re-evaluated across the candidate ecosystems to clarify and further contrast the interrelations among the conceptual dimensions of the identified characteristics.

Step 3 – External Validation: In the third and final step, we applied the taxonomy to well-known, variability-intensive ecosystems frequently cited in the SPL and clone-and-own research domains to test its external boundaries. This external validation helped to distinguish genuinely CDV-inducing traits from those that only superficially resemble them, thereby sharpening the conceptual definitions established in the previous step. As anticipated, some individual (sub)characteristics also appeared in these classical variability paradigms. This partial overlap was expected, given that our broader research scope still resides within the domain of variability-intensive systems. However, as we will show later in Section 5, these classical variability paradigm representatives do not exhibit all traits that collectively define CDV, underscoring the distinctiveness and novelty of the CDV paradigm. Recognizing these partial overlaps was essential for refining the taxonomy's discriminatory power and ensuring that the identified characteristics capture CDV as a systemic phenomenon rather than an arbitrary set of overfitted traits.

We repeated the three steps iteratively, each cycle yielding incremental refinements and sharper delineations of the emerging characteristics until the taxonomy reached a stable form. Across all three steps, we continuously resolved ambiguities and disagreements among the authors to reach consensus, while also integrating feedback from domain experts to iteratively refine and validate the emerging taxonomy. The resulting taxonomy of constituting CDV characteristics is presented in the remainder of this section. Acknowledging the pioneering nature of this work, we abstain from claiming completeness of this constituting characteristics. Yet we believe that the employed

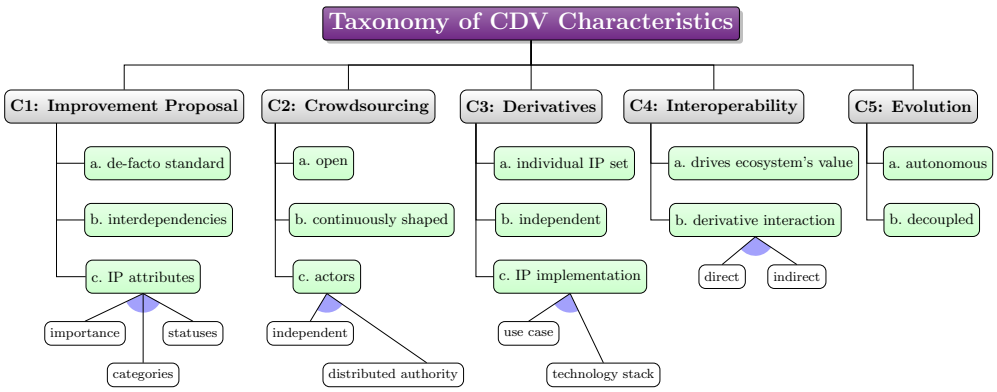


Figure 5: Taxonomy of CDV characteristics.

iterative three-step process ensures conceptual soundness and correctness. In contrast to the short and informal sketch presented in our preceding paper [43, Fig. 3], the taxonomy presented in this work offers a more fine-grained and validated account of the traits that give rise to CDV. Particularly, we emphasize the key sub-characteristics composing each characteristic and summarize the hierarchy in Figure 5: five core characteristics partitioned into 13 sub-characteristics (green boxes), which may be further refined by supplementary attributes (bottom uncolored boxes).

C1 – Improvement Proposals

"There exists a de-facto standard that defines how an ecosystem shall operate using a set of improvement proposals (IPs) that can have dependencies, varying levels of importance, and undergo different states."

The phenotype of a software ecosystem exposing CDV should be defined by a set of IPs, exhibiting the notion of specification documents. To fulfill C1, a given (sub)set of these IPs should represent **(a) a de-facto standard** that is accepted by the majority of the actors in the ecosystem. Further, these IPs are designed in such a way that they share **(b) interdependencies** among each other and possess **(c) attributions** on level of importance, evolve through different statuses, and target different layers or categories. While verifying the existence of attribution mechanisms and interdependencies is relatively straightforward, determining the presence of a de-facto standard is more ambiguous. We judge the latter using proxies such as widespread implementation across derivatives, consistent references in technical discourse, and our own internal review to assess whether a consensus exists.

C2 – Crowdsourcing

"This de-facto standard of the ecosystem is open and continuously shaped by independent actors with distributed authority."

Crowdsourcing ensures that ecosystem evolution is driven by community contributions rather than by centralized control. This requires the ecosystem to be **(a) fully open-source**, allowing diverse contributors to influence its progression and evolution. Beyond source code, this openness must extend to documentation, design, and other artifacts essential for ecosystem understanding and operation. To ensure the ecosystem's continued vitality and a truly community-driven character, we further require **(b) continuous development** by **(c) independently acting participants without central authority**, including IP contributors, derivative developers, end users, and researchers. In practice, these criteria are often met when contribution is genuinely open. However, we explicitly define C2 to exclude ecosystems that, despite being nominally open-source, remain effectively closed. For instance, those controlled by a single company where only employees contribute, or where the code is publicly visible but external contributions are not permitted.

C3 – Independent Derivatives

"Developers choose a set of IPs from which they implement independent derivatives using different technology stacks and targeting different use-cases."

Arguably the most intuitively identifiable CDV characteristic concerns the presence of diverse software applications – here denoted as derivatives – that thematically reside and interact within the same ecosystem. Developers retain full autonomy in **(a) selecting which IPs to implement** and in deciding how to realize them technically. While their development proceeds **(b) independently**, they share a common orientation toward the underlying IP catalog, which constitutes the ecosystem's de-facto standard. The notion of independence here also implies that derivatives are not tightly coupled: they do not rely on the same developers or on one another for functionality, interfaces, or updates. As illustrated by the example of Bitcoin, such derivatives are **(c) built using different technology stacks and/or target distinct use cases**. In contrast, C3 is insufficiently fulfilled in ecosystems where a substantial number of derivatives are implemented through shared components, or exhibit homogeneity in technology and purpose.

C4 – Interoperability

"The ecosystem's value and flourishing substantially depends on and encourages direct or indirect derivative interaction."

While many ecosystems emerge from crowdsourced, IP-driven development that gives rise to a variety of derivatives, interoperability among these is often incidental or altogether unimportant. Frequently, such derivatives are designed to operate in isolation, with no requirement for interaction. In contrast, CDV ecosystems are defined by interoperability as a core trait. Ecosystems fulfilling C4 share a common interest in maintaining compatibility among derivatives, as the **(a) ecosystem's overall value** – and the incentive to engage with it – fundamentally depends on this interoperability. Such **(b) interaction between derivatives** may be either direct (e.g., a Nostr client

communicating with relays) or indirect (e.g., different Bitcoin wallets interpreting the same blockchain data).

C5 – Decoupled Evolution

"The de-facto standard, its feature specification, and the derivatives evolve autonomously and detached from each other while following their own life cycles."

The fifth and final constituting characteristic of CDV ecosystems concerns their unrestrained evolution, which can be decomposed into two sub-characteristics. First, evolution is **(a) autonomous**, meaning that the de-facto standard, individual IP catalogs, and derivatives evolve with little to no centralized coordination. We consider this trait not fulfilled when a central organization with authoritative control is present, as its mere existence may disproportionately influence the direction of ecosystem evolution. Second, evolution is **(b) decoupled**, indicating that these elements progress independently, each following its own life cycle and development trajectory. An illustrative example is BIP39 [58], which – despite retaining "proposed" status until late November 2024 – had already been widely adopted by multiple derivatives for years.

5 Fulfillment of CDV Characteristics in Various Variability Paradigms

The taxonomy constructed in the previous section (cf. Figure 5) establishes a shared vocabulary and conceptual clarity for both researchers and practitioners, and also serves as a constructive tool for systematically evaluating and comparing ecosystems. In this section, we apply this taxonomy to classify ecosystems that exhibit different variability paradigms, following the process outlined in Section 5.1. With this, we aim, on the one hand, to illustrate the breadth and diversity of the CDV ecosystem landscape. On the other hand, we showcase how closely related other variability paradigms (i.e., SPL and clone-and-own) are to CDV, and also how they differ. Our classification thus reveals concrete opportunities to transfer methods and insights between CDV and these established paradigms, fostering cross-paradigm innovation and research synergies. While our main findings are presented in Table 1 and explained in detail for each ecosystem in Sections 5.2 to 5.5, we provide a summary of our classification in Section 5.6.

5.1 Taxonomy-Based Classification Process

For the classification process, we reutilized the set of ecosystems accumulated during the three-step taxonomy derivation process described in Section 4. Specifically, the candidate ecosystems identified in 'Step 1 – Discovery' formed the core of this set, as they were originally selected for their high likelihood of exhibiting CDV-related dynamics. We complemented this set with the prominent SPL and clone-and-own ecosystems already used in 'Step 3 – External Validation' of the taxonomy derivation process. Together, this selection provides a diverse and balanced basis for classification, spanning from archetypal CDV cases to classical variability-intensive systems.

For each ecosystem, we rate each of the sub-characteristics in binary terms, and mark its outcome as either fulfilled (✓) or not (-). In cases where a sub-characteristic is further refined by supplementary attributes, it is considered fulfilled if at least one attribute applies (logical OR, cf. Figure 5). The resulting scores are summarized in Table 1. We emphasize that CDV affiliation unfolds along a continuum rather than representing an absolute classification. Our presented taxonomy therefore serves as an analytical instrument for identifying the presence of CDV-inducing traits rather than as a diagnostic threshold. Since CDV affiliation is not an all-or-nothing condition, we deliberately abstain from defining a rigid cut-off. For presentation purposes, however, we grouped the ecosystems in Table 1 according to their observed tendency: ecosystems displaying a pronounced concentration of CDV characteristics are grouped using the paradigm label *CDV*, while the remaining ones are classified under their classical variability paradigms (*SPL* and *clone-and-own*) or assigned to a residual category (*Others*).

As with the taxonomy of CDV characteristics itself (cf. Figure 5), we do not claim completeness regarding the ecosystems presented in Table 1. Undoubtedly, additional and less prominent software ecosystems may exist that fall within or between the identified paradigm groups, depending on their specific sub-characteristic manifestations. This is consistent with – and indeed intended by – our view of CDV as a

Table 1: Assessment of CDV characteristics in representative ecosystems.

Paradigm	Ecosystem/Project	C1			C2			C3			C4		C5	
		a	b	c	a	b	c	a	b	c	a	b	a	b
CDV	Bitcoin [26, 27]	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	Lightning [63–65]	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	Ethereum [66, 67]	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	-	✓
	Nostr [32, 33]	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	Tor Protocol [30, 31]	✓	✓	✓	✓	✓	✓	✓	-	✓	✓	✓	-	✓
	IPFS [28, 29]	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	-	✓
SPL	Linux Kernel [68, 69]	✓	✓	-	✓	✓	✓	-	-	-	-	-	-	-
	Eclipse [70]	-	✓	✓	✓	✓	✓	✓	✓	✓	-	-	-	-
	BusyBox [71]	✓	✓	-	✓	✓	✓	-	-	-	-	-	-	-
Clone-and-Own	Apo-Games [72]	-	-	-	-	-	-	✓	✓	✓	-	-	-	✓
	Marlin Forks [73]	-	-	-	✓	✓	✓	✓	✓	-	-	-	✓	✓
	Health Watcher [74]	-	-	-	-	-	-	✓	✓	-	-	-	-	✓
Other	Home Assistant [75]	-	-	-	✓	✓	-	-	✓	✓	-	✓	-	✓
	Python [76, 77]	✓	✓	✓	✓	✓	-	✓	✓	✓	-	-	-	✓

continuum. By focusing on correctness rather than completeness, the presented classification yields strong evidence that CDV constitutes a distinct and novel paradigm that deserves dedicated academic attention.

5.2 CDV Ecosystems

We first examine ecosystems that, according to our taxonomy, most prominently exhibit CDV characteristics as naturally emerging phenomena and analyze how these traits manifest across them. In doing so, we deliberately adopt an observational stance, describing the phenomena as they appear in practice, similar to early studies in variability-intensive systems research. For brevity, we use the term "*CDV ecosystem*" to denote a software ecosystem that largely fulfills the constituting CDV characteristics and thereby exemplifies the CDV paradigm, with its specific peculiarities and problems. Building on our motivating example Bitcoin, we extended the analysis to the broader set of candidate ecosystems accumulated during 'Step 1 – Discovery' of the taxonomy derivation process (cf. Section 4). This includes adjacent systems within the electronic money domain, allowing us to explore the generalizability of CDV phenomena under related conditions. In addition, we incorporated examples from outside this domain that, based on our research, showed a high likelihood of exhibiting CDV characteristics.

5.2.1 Bitcoin and Lightning

In Section 3, we already discussed Bitcoin [26, 27] in detail, acting as guiding example for the observed CDV phenomena and as foundation for generalizing its defining characteristics. Another prominent example of such an ecosystem is the Lightning Network [63], which is a second-layer protocol built on top of Bitcoin for enabling instant off-chain payments. Despite being closely related to Bitcoin, Lightning exhibits its

own IP catalogs, specifically through the *Basis of Lightning Technology* (BOLT) [64] and *Bitcoin Lightning Improvement Proposals* (bLIPs) [65]. Later, a third IP source evolved called *Lightning URL Documents* (LUDs) [78], which further extends the two catalogs specifically towards URL specifications to streamline inter-derivative communications.

Similarly to Bitcoin, the derivative spectrum in Lightning features multifaceted and independent implementations (3/3 in C3) such as, for instance, Lightning Network Deamon (LND) [79] written in `Go`, Eclair [80] written in `Scala`, and Core Lightning (CLN) [81] written in `C`. Besides their heterogeneous technology stack, these derivatives also focus on different IP sets (3/3 in C1), which has even led to the introduction of *feature flags* according to BOLT9 [82] in the form of bit vectors to maximize interoperability (2/2 in C4). Yet, BOLT9 states that *"[s]ome features [...] became so widespread they are* ***`ASSUMED`*** *to be present by all nodes"*, underpinning the existence of a de-facto standard (3/3 in C2). Likewise, the LUDs catalog [78] states, that *"[e]ach new LUD may be implemented by some wallets and not others, some services and not others, but they should still maintain compatibility at all times"*. With the absence of a central authority as in Bitcoin, the Lightning ecosystem and its components evolves autonomously and detached from each other (2/2 in C5). Overall, we find the Lightning ecosystem to show all of our 13 sub-characteristics.

5.2.2 Ethereum

Ethereum [66] arguably represents the second most prominent blockchain ecosystem after Bitcoin, distinguished by its programmable smart contract functionality and support for decentralized applications (dApps). Unlike Bitcoin's primary focus on peer-to-peer value transfer, Ethereum operates as a distributed computing platform that enables developers to deploy arbitrary code through smart contracts. The ecosystem comprises diverse components including blockchain clients, development frameworks, wallets, and dApp platforms, all coordinated through *Ethereum Improvement Proposals* (EIPs) [67] that govern protocol evolution and feature specifications.

C1 is fully satisfied (3/3) through EIPs [67] that are segmented in categories like *core* or *networking*, and possess statuses like *draft*, *final*, or *withdrawn*. The designated "Required" field in EIPs attests interdependencies, as it is the case with, e.g., EIP4938 [83].

Like Bitcoin and Lightning, the Ethereum ecosystem is open and actively developed. While the Ethereum Foundation holds a disproportionately dominant position and exercises notable authority [84], contributions remain open, and independent developers and organizations actively shape the ecosystem. We therefore assign C2 a score of 3/3.

As in Bitcoin and Lightning, the Ethereum ecosystem features a large number of derivatives, built with different technologies, targeting different use cases and thus also emphasis different sets of EIPs. This includes, for instance, the Ethereum client Geth [85] written in `Go`, the browser-extension wallet MetaMask [86] written in TypeScript, and the Non Fungible Token (NFT) marketplace protocol OpenSea [87] written in `Solidity`. Consequently, C3 is fully satisfied with a score of 3/3.

Like Bitcoin and Lightning, Ethereum's token transfer protocol incentivizes seamless interoperability: clients synchronize state, wallets interact with smart contracts, and dApps communicate across the network. Thus, C4 is fully satisfied with a score of 2/2.

Finally, the Ethereum ecosystem evolves in a largely decoupled manner. Contributions to EIPs are not coordinated with the evolution of concrete implementations in the derivative spectrum, as these artifacts are maintained by independent individuals or teams. Yet, the influential role of the Ethereum Foundation constrains the full autonomy of this evolution. Thus, C5 is partially satisfied with a score of 1/2.

5.2.3 Nostr

Nostr [32], short for 'Notes and Other Stuff Transmitted by Relays', is a decentralized protocol based on a publish-subscribe communication model secured through asymmetric cryptography. By design, it maintains a lightweight architecture composed of only two core components: clients, which generate and consume events, and relays, which receive and disseminate them. Although it has primarily gained traction as a protocol powering decentralized social media alternatives to platforms like Twitter/X, its architecture is broadly applicable to other use cases, including microblogging, collaborative editing, video platforms, and identity systems. Some interpretations even propose that Nostr may serve as a *social layer* within the Open Systems Interconnection (OSI) model [88], offering the possibility of integration with diverse application architectures.

As mentioned earlier, the Nostr ecosystem is defined and progressed on using *Nostr Implementation Possibilities* (NIPs) [33], which function as community-driven specifications for protocol extensions and behaviors while also functioning as de-facto standards. These NIPs also exhibit interdependencies as, for instance, NIP46 [89] depends on NIP44 [90]. We also record that the NIPs are attributed with statuses like *draft*, *final*, or *unrecommended*. Thus, C1 is fully satisfied with a score of 3/3.

Although the initiator of the Nostr protocol is well-known [91], the ecosystem today's dynamism is largely crowdsourced. The project is fully open-source and – according to its GitHub statistics [33] – vividly developed by a wide range of actors. Ergo, we assign a score of 3/3 to C2.

Nostr [33] is a decentralized social network protocol that allows users to publish and subscribe to messages. The variety of Nostr derivatives is vast, with implementations ranging from simple command-line clients to complex web applications. As of the time of writing, for example, around 1'000 relays were recorded online, operating on 79 different software stacks [92]. In fact, the community attempts to maintain oversight of this heterogeneous derivative landscape using handcrafted indexes and catalogs [93, 94]. Consequently, C3 is fully satisfied with a score of 3/3.

As in Bitcoin, Lightning, and Ethereum, the Nostr ecosystem naturally focuses on interoperability, driving its value. Interaction between clients and relays is a core feature of the protocol, enabling users to publish and subscribe to messages of all kinds. Thus, C4 is fully satisfied with a score of 2/2.

Finally, the Nostr ecosystem evolves autonomously and detached from each other. The contributions to NIPs are not orchestrated with development progression in,

e.g., the aforementioned derivatives, nor vice-versa as these artifacts are developed by independent teams or individuals. We therefore deem C5 as fully satisfied with a score of $2/2$.

5.2.4 Tor Protocol

The Onion Router (Tor) [30, 31] is a privacy-focused overlay network designed to enable anonymous communication over the Internet. It achieves this by routing traffic through a series of nodes using layered encryption, obfuscating the source, destination, and content of network packets. Tor is primarily known for powering privacy-preserving web browsing through the Tor Browser [95], but its underlying protocols and network are also leveraged in applications like onion services and secure communications infrastructure.

As part of its architectural evolution, the Tor ecosystem relies on a set of structured design documents known as Tor Design Proposals (TorDPs) [31], defining the de-facto standard. As in the previously discussed ecosystems, TorDPs possess statuses (e.g., *open*, *accepted*, *needs-revision*) and may reference other proposals. For instance, TorDP324 [96] depends on TorDP289 and TorDP325, or TorDP291 [97] has superseded TorDP236. We deem C1 therefore fully fulfilled with $3/3$.

While Tor is open-source and public contributions are technically possible, the development and governance of the protocol and related software are largely coordinated by The Tor Project itself, a non-profit organization [98]. This includes core Tor developers, employed maintainers, and project leads who guide most architectural decisions and manage the life cycle of proposals. Nonetheless, input can come from the wider community, with contributors acting and contributing independently of this central authority. We thus grant C2 a score of $3/3$.

The Tor ecosystem features components across multiple layers, including the `C`-based Tor daemon [30] and the Tor Browser [95] built with `JavaScript` and `C++`. While these projects differ in technology and main functionality, most are developed under the coordination of The Tor Project [99], and fully independent derivatives are scarce. We therefore assign C3 a score of $2/3$.

As in Nostr, interoperability is essential for a network protocol like Tor, where clients must communicate with the distributed relay infrastructure to enable core functionality such as traffic routing. Mechanisms for backward compatibility and feature negotiation are built into the protocol [100, 101]. Thus, C4 is fully satisfied with a score of $2/2$.

Although The Tor Project coordinates much of the protocol development, parts of the ecosystem – such as onion services (Tor-based services accessible only via `.onion` addresses) and bridges (unlisted relays used to bypass censorship) – evolve with their own timelines and maintainers. While a few derivatives exist outside the organization's core projects, the ecosystem remains more centrally coordinated than, e.g., Bitcoin or Nostr, which is why we do not credit the autonomous evolution sub-characteristic. The absence of strict top-down integration across derivatives, however, implies a degree of decoupling in the evolution trajectory. Accordingly, C5 is partially fulfilled with $1/2$.

5.2.5 IPFS

IPFS [28, 29], short for *InterPlanetary File System*, is a decentralized protocol for content-addressed file sharing across a peer-to-peer network. As Nostr, IPFS minimizes architectural complexity by relying on simple primitives. Nodes store, retrieve, and replicate data based on cryptographic hashes rather than centralized URLs. While originally conceived as a more resilient and decentralized alternative to the traditional web, IPFS has since been adopted in diverse contexts, including static site hosting, blockchain data availability, and archival storage. It is increasingly viewed as a core building block for decentralized application infrastructure [102].

Although the IPFS ecosystem defines its architecture and core principles primarily through conventional documentation, IPIPs [29] nonetheless serve as the formal vehicle for proposing and tracking protocol-level changes and new features. IPIPs exhibit structured metadata such as status labels (*draft*, *accepted*, *final*, *withdrawn*) and inter-proposal references. We therefore assign C1 a score of $3/3$.

IPFS is fully open-source and technically open to collaborative development across multiple organizations and individual contributors. While development is primarily coordinated by Protocol Labs [103], which retains strong influence over the protocol's direction, individuals can still contribute independently and shape parts of the ecosystem. We therefore assign C2 a score of $3/3$.

The IPFS ecosystem features numerous independent implementations and derivatives built with different technology stacks and targeting various use cases. Renowned implementations include `kubo` [104] written in `Go`, or `helia` [105] written in TypeScript. Beyond these, numerous other specialized projects exist with each serving different user needs and deployment scenarios [106]. We therefore assign C3 a score of $3/3$.

Interoperability is central to IPFS's design philosophy, aiming to enable seamless data exchange across diverse implementations and platforms. Accordingly, interaction among derivatives clearly drives the ecosystem's value and growth, justifying a C4 score of $2/2$.

The IPFS ecosystem exhibits decoupled evolution, with various tools and implementations evolving on their own timelines and objectives. As with Ethereum, while the ecosystem demonstrates decoupled evolution across different implementations and use cases, the strong influence of Protocol Labs in coordinating protocol development limits the degree of autonomous evolution. We therefore assign C5 a score of $1/2$.

5.3 Software Product Lines

Software product lines (SPL) [107–109] define and implement variants in an integrated platform. The variability of the system is defined in terms of a feature model [110, 111], which defines the features of the system and how these features are related. Variants of the system are generated using a variability mechanism such as the `C`-preprocessor with its `#if` and `#ifdef` macros. The variability mechanism determines which code should be included based on the features that are selected for a variant.

There are substantial differences between SPL development and CDV. In contrast to CDV derivatives, SPL variants have a shared set of artifacts that implement the

features of the system. Variants are not changed directly but indirectly by changing the shared artifacts and generating updated versions of the variants. Lastly, SPL variants are typically meant to operate independently and address the needs of different customers or platforms (e.g., different hardware, embedded vs. general-purpose architecture). We selected SPL case studies that are well-documented in the literature, ensuring a reliable basis for contrasting them with CDV ecosystems.

5.3.1 Linux Kernel

The Linux kernel [68, 69] serves as the foundation for numerous operating systems, including various Linux distributions and `Android`. The features of the kernel are defined in terms of configuration options using KConfig [112], a custom configuration language which allows specifying feature hierarchies and dependencies. The features are primarily implemented in `C`, and the variability of the source code is defined using the `C`-preprocessor and the kernel's custom build system, KBuild. Different variants of the kernel can be generated by selecting desired features and then calling KBuild to compile the variant. We included the Linux kernel as ecosystem since it is one of the most popular subjects for product line research (e.g., [113–116]) given its practical relevance and its huge size. The number of features grows steadily and there are about 20,000 features in 2024 [117].

While the Linux kernel does not use a formalized system of improvement proposals (IPs) akin to CDV ecosystems, there exist a de-facto standard in form of configuration options. These options are explicitly documented in the KConfig configuration files with their dependencies and other relationships. New configurations options can be proposed via mailing lists, where they undergo community review and discussion before potential inclusion. Although configuration options follow an implicit lifecycle, their status is not explicitly tracked or attributed. We therefore assign C1 a score of $^2/_3$.

The Linux kernel is a continuously evolving open-source project to which anyone can contribute by submitting patches. It is under active development, and we deem contributors to act largely independently. While a degree of central authority remains with its initiator, Linus Torvalds, he does not review every submitted patch. The responsibility for decision-making is instead distributed among independent sub-groups and maintainers. We thus conclude that C2 is fully fulfilled with $^3/_3$.

As a well-established representative of SPL, the Linux kernel generates all variants from a shared set of artifacts. Consequently, the variants are fully dependent on each other regarding their implementation and evolution, leading to a C3 score of $^0/_3$.

Each variant of the Linux kernel is the foundation for a specific instance of an operating system (OS). The main value of this system does not come from interoperability between OS instances but from the general functionalities of an OS. Although interactions may occur in rare cases, such as in distributed file systems, they are not typical. We therefore assign C4 a score of $^0/_2$.

Finally, C5 is also not fulfilled ($^0/_2$) because configuration options and their implementation, from which variants are generated, always evolve together in a coordinated way.

5.3.2 Eclipse

The Eclipse IDE [70] is a plugin-based software platform that can be tailored for various purposes. For instance, it can be configured as an IDE for `Java`, `C++`, or as a framework for modeling tools. Variants of Eclipse are created by selecting desired sets of plugins, with several pre-configured distributions available that users can further customize. The project is open source and maintained by the Eclipse Foundation [118], which oversees its development. Anyone can submit change requests, report issues, or propose code changes, which are reviewed by project committers before inclusion. Additionally, it is possible for developers to create and distribute their own plugins to extend the platform's functionality. Due to its flexible architecture and plugin ecosystem, Eclipse is frequently cited in the literature as an example of a SPL (e.g., [119–123]).

Each plugin in the Eclipse ecosystem can be viewed as a separate IP: It typically documents its purpose, functionality, and metadata such as development status, supported Eclipse versions, or OS. Interdependencies between plugins may also be explicitly documented. However, the set of existing plugins does not constitute a de-facto standard. Instead, we find that they are optional extensions to the Eclipse IDE. We therefore assign C1 a score of $2/3$.

While core changes are centrally governed under the authority of the Eclipse Foundation, anyone can contribute new plugins to the ecosystem. As a result, the plugin landscape is continuously shaped, involving independent developers and organizations. We therefore assign C2 a score of $3/3$.

On the same note, the development of plugins happens independently and developers can freely choose which new functionality they want to implement. Although most plugins are written in `Java` and must integrate with Eclipse's extension points, they target a wide range of different use cases. Since our criterion is satisfied by either technological diversity or diverse use cases, we assign C3 a score of $3/3$.

Eclipse variants (i.e., installations with different plugin sets) generally do not need to interoperate. The ecosystem's value lies in supporting different programming and modeling languages, not in enabling interaction between Eclipse installations. Accordingly, C4 is rated at $0/2$.

Similarly, C5 is not fulfilled ($0/2$), as plugins combine both the specification and implementation of an IP, causing them to evolve together.

5.3.3 BusyBox

The developers of BusyBox [71] refer to it as the "Swiss Army Knife of Embedded Linux". It is a collection of Unix utilities (e.g., `cat`, `ln`, and `ls`) compiled into a single binary after configuration. Commonly used in embedded Linux systems, BusyBox can be tailored to specific architectures and hardware. It employs the same configuration and variability mechanisms as the Linux kernel,namely KConfig and KBuild, to define features and generate variants. Given that it is not as immense in size as the Linux kernel and still manageable (cf. [124]), it is frequently referenced in SPL research (e.g., [125–129]) particularly for variability analysis. Due to their close relationship, the

development of BusyBox and the Linux kernel are highly similar. Thus, the scores with respect to the different characteristics of CDV are also the same.

Busybox also has a de-facto standard in form of configuration options with inter-dependencies, but no attributions (C1 $^2/_3$). BusyBox is also open-source and under continuous development, to which anyone can contribute by submitting patches. We therefore assign C2 a score of $^3/_3$. Its variants are generated from a common set of artifacts and fully dependent on each other (C3 $^0/_3$). Each variant of BusyBox is a toolbox tailored to the requirements of a specific platform. The main value of Busybox comes from the utilities that it provides to a system. We are not aware of any interoperability between variants (C4 scored with $^0/_2$). Lastly, same as for Linux, C5 is not fulfilled ($^0/_2$) because configuration options and their implementation always evolve together.

5.4 Clone-and-Own Projects

The term clone-and-own refers to the practice of cloning and adapting existing software variants to quickly create new ones [12, 13, 130]. The process usually starts with a single software variant that is implemented without planning for future variants. Once the need for a different variant of the system arises (e.g., due to the demands of a new customer), developers clone the variant by duplicating all its software artifacts. The thereby created clone is then adapted by changing the duplicated artifacts, for example, by removing undesired functionality.

The only similarities between clone-and-own and CDV are the notion of independent derivatives (variants), and that they can evolve autonomously and detached from each other. However, even in this regard, clone-and-own variants are less independent than CDV derivatives, because clone-and-own variants are cloned from each other and use the same technology stack, while CDV derivatives are fully independent and may use any technology stack. Furthermore, there is no de-facto standard based on which the variants are implemented, and variants can undergo arbitrary adaptations. Clone-and-own variants are also not intended to be interoperable as they represent unique customizations of a system for different purposes. To analyze how CDV characteristics contrast with more traditional forms of variability, we selected representative case studies of clone-and-own development based on their prominence in the literature and the availability of detailed information.

5.4.1 Apo-Games

The Apo-Games are a series of games developed by Dirk Aporius [72] using clone-and-own. The games were developed in `Java` and `Android`. Each game developed before 2013 has been created by cloning and adapting artifacts of its predecessors, while there was a break in this process due to a framework migration that required a new implementation for newer games. Together with Dirk Aporius, Krüger et al. [131] selected 20 `Java` and 5 `Android` variants as a realistic case study for the reverse engineering of domain knowledge (e.g., knowledge about features and how they are implemented). This case study has been used by several other works on clone-and-own development (e.g., [132–136]). While the source code of the selected 25 variants was

made available, the remaining variants and the original projects of the 25 variants are not available publicly.

As is typical for clone-and-own projects, Apo-Games does not have an explicitly documented set of IPs, or anything similar (0/3 in C1). While some variants were made publicly available in form of the case study mentioned above, the development of Apo-Games is performed closed-source by Dirk Aporius (0/3 in C2).

The variants of Apo-Games were developed independently of each other by cloning and adapting an existing variant. For each variant, Dirk Aporius could freely choose which features of the cloned variant to retain and which new features should be added. Each variant is the implementation of a different game, which we consider as variants targeting distinct use cases. Thus, we score C3 with 3/3. However, the Apo-Games variants are standalone games that have no form of interoperability, leading to a score of 0/2 in C4.

Apo-Games does not fulfill the sub-characteristic of autonomous development because there is a single developer with sole authority. Nevertheless, the evolution of variants is decoupled and each variant had its own lifecycle and development trajectory. Thus, we score C5 with 1/2

5.4.2 Marlin Forks

Marlin [73] is an open source firmware for 3D printers written in `C++` and `C`. The Marlin projects aims at supporting various boards and machine configurations and is highly customizable. Several research works (e.g., [130, 137–140]) focus on a subset of the thousands of forks of Marlin, which are considered a practical case study for clone-and-own development [130]. While many forks are social forks that are created to contribute to the original project, the majority of forks diverge from the original constituting separated variants [130]. Here, cloning is done by independent developers or groups of developers that simply fork the original project on GitHub.

There is no common, de-facto standard for the forks of Marlin. Each fork may implement arbitrary new functionality without specifying or documenting this functionality. Thus, we attribute C1 with 0/3 Marlin and its forks are publicly available and anyone can contribute to the ecosystem, either by creating a new fork (i.e., a new variant), or by contributing changes to one of the existing variants. The ecosystem therefore undergoes continuous development and is shaped by independent actors. We score C2 with 3/3.

Typical for clone-and-own variants, the forks of Marlin are implemented independently of each other and developers can, in principle, freely choose which configuration options (IPs) to implement. However, in the case of Marlin, all variants target the same use case (i.e., 3D printer firmware), and have a shared technology stack due to cloning. Thus, we score C3 with 2/3 The printers and their firmware do not have to interact or interoperate (C4 0/2).

The evolution of Marlin forks is generally autonomous and decoupled. Each fork may have its own developer or group of developers and forks may heavily diverge from the mainline. Forks have their own lifecycle that is independent of other forks. Each

fork may alter how a specific IP (configuration option) is implemented and the de-facto standard and the derivatives are essentially decoupled, resulting in a C5 a score of 2/2.

5.4.3 Health Watcher

Health Watcher [74] was developed to improve the service quality of health care institutions by allowing the public to submit health complaints. For example, users could report restaurants, prompting investigations by the responsible authorities. Health Watcher has been used as a case study in several research works on clone-and-own development (e.g., [141–145]). The case study consists of 10 variants representing different releases of the system, each involving multiple changes such as feature additions or refactorings. Although these 10 variants are available online, we were unable to locate the original project and found no evidence that it was developed as an open-source initiative.

There is no explicit specification of IPs (or features) and there is no de-facto standard for the different variants of Health Watcher. Thus, C1 is not fulfilled (0/3). As we were not able to locate the original project in which Health Watcher was developed, we have to take a conservative stance and assume that it was not developed through crowdsourcing. There likely was a central authority that oversaw the developed Health Watcher and the different variants. Thus, C2 is also not fulfilled (0/3).

The variants of the Health Watcher are different releases of the system. We consider them independent, as each variant could, in principle, completely change which functionality is offered and how it is implemented. Similar to other clone-and-own projects, the variants share a common purpose and technology stack. Thus, we score C3 with 2/3. Yet, Health Watcher variants have no interoperability, because they are merely different releases of the same system, not intended to interact (C4 0/2).

Lastly, we consider that the autonomous evolution of the variants is not fulfilled, because they were likely developed by the same central authority. Their evolution was decoupled, however, as they evolved after one another. Thus, we score C5 with 1/2.

5.5 Other

To broaden our perspective and provide a counterbalance to the previously discussed domains, we included ecosystems from more distant areas that appeared promising for exhibiting CDV characteristics. We examine Home Assistant from the IoT domain, where interoperability is an active area of research [146–152]. Additionally, we include Python as a widely used programming language, that also uses IP-driven methodologies for its language evolution.

5.5.1 Home Assistant

Gaining increasing popularity in recent years, Home Assistant [75] is an open platform for (smart) home automation and IoT device management. Its vendor-agnostic architecture enables users to retain control, avoiding ecosystem lock-in while facilitating flexible, self-authored automation across heterogeneous IoT environments.

While Home Assistant has extensive documentation and a structured development process, it does not rely on formal improvement proposals (IPs) in the traditional sense. Instead, feature development and architectural decisions are coordinated through GitHub issues, pull requests. The lack of a formal IP catalog with statuses and interdependencies means C1 is not fulfilled, resulting in a score of 0/3.

Home Assistant is fully open-source and actively maintained by a vibrant community of contributors. However, the project is heavily influenced by Nabu Casa [153], the company founded by Home Assistant's creator, which provides centralized direction and coordination. This centralized influence limits the distributed authority aspect of crowdsourcing, leading to C2 score of 2/3.

The Home Assistant ecosystem does not support individual IP set selection, as it lacks a dedicated IP catalog. Nonetheless, it features a range of independent extensions and tools, notably through shadow platforms such as the Home Assistant Community Store (HACS) [154, 155], which index third-party UIs, automation engines, or reverse-engineered device integrations. Although deploying and maintaining such independently developed derivatives appears to be gradually becoming more challenging, they nonetheless exhibit diversity in both technology stacks and use cases. We therefore assign C3 a score of 2/3.

While direct interoperability between derivatives remains limited and is frequently cited as a challenge within the Home Assistant ecosystem [156], we argue that indirect interactions do occur through automation workflows. For instance, integrations can be chained to trigger complex sequences across heterogeneous devices. This form of functional interoperability contributes to the ecosystem's value, albeit in a constrained manner, as most derivatives retain their utility even when operating in isolation. We therefore assign C4 a score of 1/2.

Evaluating the evolutionary characteristic, we see autonomy not fulfilled in the Home Assistant ecosystem. As with previously analyzed ecosystems, a dominant organization (Nabu Casa) plays a central role in directing the platform's development and roadmap. Furthermore, evolutionary steps are predominantly initiated by external hardware manufacturers who release new devices that the community subsequently integrates, reflecting a unidirectional evolutionary flow. In contrast, we see decoupling fulfilled, as community-driven add-ons and custom integrations can evolve independently of the core platform and demonstrate a degree of organic growth. We therefore assign C5 a score of 1/2.

5.5.2 Python

As a widely used, interpreted language with decades of active development, Python [76] serves as a worthwhile case study to assess CDV properties. Its ecosystem extends far beyond the reference interpreter (CPython), comprising alternative runtimes, a comprehensive standard library, and over 650,000 third-party packages published on the Python Package Index (PyPI) [157].

The de-facto standard of Python and its library is formalised through *Python Enhancement Proposals* (PEPs) [77], that cross-references to related proposals (e.g., PEP654 depends on PEP622). They also carry structured metadata such as type

(standards-track, informational, process) and status (*draft*, *accepted*, *rejected*). Hence, C1 is fully satisfied with 3/3.

Python is open-source and continuously shaped through publicly visible contributions and reviews. However, the language's technical direction and maintenance is influenced by the Python Software Foundation (PSF) [158] and its elected Steering Council (defined in PEP13 [159]), which oversees the acceptance of PEPs. We therefore assign C2 a score of 2/3.

Besides the reference interpreter CPython other independent derivatives exist such as MicroPython [160] (written in `C`) or RustPython [161]. These projects are maintained by independent teams and may implement only a subset of the available PEPs. For example, MicroPython provides a dedicated documentation page on feature differences as it *"implements Python 3.4 and some select features of Python 3.5 and above"* [162]. Consequently, we assign C3 a score of 3/3.

Python derivatives function as standalone interpreters, with generally no practical expectation of cross-runtime execution or artifact sharing. Unlike the previously treated protocol-based ecosystems, Python's value does not substantially rely on such interaction. Consequently, C4 is rated 0/2.

Python derivatives evolve on independent timelines, e.g., MicroPython and RustPython maintain separate release cycles that diverge from CPython's annual cadence, demonstrating decoupled evolution. However, core language development remains coordinated by the PSF and its Steering Council, limiting the autonomous evolution of the overall ecosystem. We therefore assign C5 a score of 1/2.

5.6 Summary

The main findings of our analysis in this section are presented in Table 1. CDV ecosystems consistently fulfill all or nearly all sub-characteristics of our taxonomy from Figure 5, with only Ethereum, Tor and IPFS missing one or two of them. In contrast, ecosystems from other paradigms (SPL, Clone-and-Own, and Other) show significantly lower fulfillment. However, given the limited number of representative ecosystems, we advise caution in assuming that other ecosystems within the same paradigm would exhibit identical characteristic profiles.

A key insight emerging from our analysis is that none of the non-CDV ecosystems exhibit derivative interactions that give rise to a shared interest in interoperability (C4) that is essential to the ecosystem's overall value. In contrast, CDV ecosystems are characterized by such interactions, often incentivized by design, fostering a sustained commitment to compatibility despite notable heterogeneity in purpose and technology stacks. This makes derivative interaction – and the resulting drive for interoperability – a defining and potentially exclusive trait of CDV ecosystems. Moreover, the autonomous and decoupled evolution of IPs, derivatives, and de-facto standards appears to be a necessary condition for sustaining these interaction patterns at scale. CDV ecosystems exemplify this, whereas ecosystems from other paradigms exhibit tighter coupling and a more linear relationship between specification and implementation, limiting the versatility of evolutionary trajectories. This interplay of interoperability and autonomy may thus serve as a useful indicator for identifying or predicting CDV-like dynamics in emerging systems.

6 Emerging Problems

We identified several generalizable challenges faced by key actors in Community-Driven Variability (CDV), including IP maintainers, derivative developers, and end users. We focus on those that transcend classical variability-intensive systems and were confirmed in our interviews with Bitcoin experts. In the following, we present an exemplary selection, enriched by concrete anecdotes from the Bitcoin ecosystem. Note, however, that the listed problems are not confined to Bitcoin but also inherent to other ecosystems due to the fundamental characteristics of CDV.

P1 & P2 – Missing overview of proposal and derivative spectrum: Due to the dynamics imposed by characteristics C2-C5, communities typically lack an overview of the entire ecosystem and its evolution. Consequently, involved actors lack orientation for guiding their decisions within the ecosystem. This missing overview is felt on both levels: the proposal spectrum (P1), and the derivative spectrum (P2). Realizing the need for an overview, the Bitcoin community already created a number of websites that monitor [41], compare [39, 42, 163], or suggest [40] derivatives. These efforts are largely handcrafted and ad-hoc, reflecting a highly manual process that highlights both the richness of existing variability and the need for more systematic management.

P3 – IP change impact assessment: The actors (C2) in the ecosystem face challenges during suggesting and updating IPs (C1), such as avoiding unforeseen side effects and change impact assessment (C4). For example, although on-boarding developer guidelines exist in Bitcoin [164], resources that document the interrelations between BIPs or their perceived feature impacts are missing. As of today, contributors must manually trace dependencies, cross-checking IPs and inferring potential side effects, or rely on experts with tacit knowledge of the ecosystem.

P4 – Misalignment of proposal and derivative spectrum: There is a common interest to avoid a misalignment (C5) of derivatives and the proposal spectrum. Yet, developers (C3) lack necessary guidance for alignment, while end users are unable to verify it, undermining trust in derivatives (C4) and in the ecosystem. This lack of guidance is exemplified in Electrum avoiding BIP39 [62], whereas Sparrow *"tries wherever possible to adhere to commonly accepted standards [to] have as wide an interoperable"*. [57]

P5 – Determining interoperability of derivatives: The shared interest in interoperability (C4) forces developers and end users to be aware of potential restrictions of derivative interactions. A lack of interoperability can lead to immense damage, such as permanent financial losses due to wallet recovery issues [38, 165] or incorrectly mined blocks [166]. As mentioned earlier, some communities already introduced partial solutions for this problem, e.g., *feature flags* [82], a handshake, that tests what features the other derivative implements prior to actual interaction. However, users could place more trust into a more rigorous procedure, that is formally derived from and enforced through an ecosystem's variability model. Currently, interoperability between derivatives largely depends on cumbersome manual testing or becomes apparent only through user reports of experienced issues.

P6 – Ecosystem fork: The independent evolution of proposals and derivatives (C5) can lead to complex phenomena: As some IPs are embraced by the whole

community, others may be rejected by a part of the community (C3). This can lead to a split within the ecosystem into fractions or a complete detachment, as sub-communities drift further and further apart. Ultimately, such detachments provoke yet another variability source for both IPs (C1) and derivatives (C3), catalyzing the severity of P1-P5. In Bitcoin and related domains this phenomenon is referred to as *fork* and has occurred several times in the past (e.g., Bitcoin Cash, Gold, SV) [167].

7 Research Vision

Having established a foundational understanding of CDV (Section 4), demonstrated its discriminative power against other variability-intensive paradigms (Section 5), and elaborated its implications as concrete problems (Section 6), this section now states our research outlook to advance this emerging space. Therewith, we aim to move from a purely observational and descriptive view of CDV as a naturally and largely unmanaged phenomenon toward a more capability-oriented perspective, where CDV exhibiting ecosystems can be systematically supported and guided. Our research vision is to develop foundations for methods supporting the continuous evolution of ecosystems exposing CDV, tackling the problems identified in Section 6. We focus on understanding and auditing its multidimensional dynamics, and on providing means for constructive, organizational and analytic quality assurance. We present our research goals, promising starting points for technical solutions with particular potential for automated approaches, and envisioned research methods and study subjects.

7.1 Research Goals

RG1 – Systematic treatment of CDV in proposal spectrum: Our first research goal is threefold. First, we aim to develop a variability modeling formalism and notation that can adequately capture CDV ecosystems and their evolution, providing a structured, explorable representation of the proposal spectrum amenable to analysis (P1). Second, we want to support the automated extraction of CDV models from various resources, with a focus on deriving variability models directly from IP collections. Third, analysis techniques shall be developed to reason about the structure and constraints of CDV models, spotting anomalous IPs and interrelations. This includes methods for differential analysis of CDV models representing different proposal spectrum snapshots, facilitating change impact analyses in the proposal spectrum (P3, P6).

Impact: Holistic modeling of a CDV ecosystem's topology fostering comprehensibility and auditability.

RG2 – Supporting cohesive evolution of proposal and derivative spectrum: Given the autonomous evolution of these two spectra, our goal is to better understand and measure their cohesion (P4). This includes providing configuration support through CDV model-guided IP selection and first cohesion assessments by, e.g., checking a given set of IPs against a CDV model. However, the major endeavor pursued with this research goal is to support tracing of IPs from the proposal to the derivative spectrum, providing a better understanding of the derivative spectrum (P2) and facilitate further change impact analyses (P3). Besides IP traceability, we aim at mining CDV models from existing derivatives, enabling comparisons with those extracted from the IP spectrum (P4) and analyzing potential drift between community forks (P6). Such mining and processing efforts should leverage automation potential to ensure replicability and sustainably support an ecosystem's development trajectory.

Impact: Streamline the evolution of ecosystems by increasing the efficiency and effectiveness of future development endeavors.

RG3 – Methodical handling of derivative interoperability impairment: Synthesizing our research goals targeting the proposal spectrum and its cohesion with the derivative spectrum, we dedicate our final research goal to address the challenges related to impaired interoperability within the derivative spectrum (P5), which boils down to handling and detecting undesired inter-derivative IP interactions. Anticipated interactions shall be documented and articulated through the CDV model, amenable to automatically validating derivatives wrt. proposal spectrum alignment (P4). Unanticipated interactions impairing interoperability shall be detected through systematic and automated IP interaction testing, which must be both effective and efficient to be accepted in practice.

Impact: Reduce the effort and complexity of proper inter-derivative feature testing, further maximizing interoperability and positive user experience.

7.2 Starting Points for Technical Solutions

In general, our technical solutions for achieving our research goals RG1-RG3 shall adopt existing variability mechanisms as far as possible, yet with radically different goals and assumptions, and without the need to adopt product-line development processes which hardly apply to the dynamics of community-driven ecosystems. While our three research goals are largely orthogonal, technical solutions for RG2 and RG3 depend on progress towards RG1, which aims to introduce a modeling formalism that serves as the backbone for various types of analysis. In the sequel, we will thus present our initial directions for addressing RG1 in more detail, while outlining higher-level considerations for RG2 and RG3.

7.2.1 Starting Points for RG1

Variability modeling formalism and proposal spectrum analysis: Inspired by classical approaches to variability modeling and problem space analysis [3], the first and most essential step towards RG1 is to develop a variability modeling formalism and notation that adequately captures CDV. While IPs in CDV ecosystems align well with the classical notion of a feature as central domain abstractions, it remains an open question whether existing feature modeling constructs are sufficiently expressive to capture the more complex nature of IPs.

In particular, IPs are not merely Boolean features as typically assumed in FODA-like feature models [168], but instead carry additional semantics, such as the word seeds of BIP39 and SLIP39 used in the BIP-based wallet creation in Sparrow (see Figure 3). We anticipate that this additional information must be considered when reasoning about ecosystem evolution and interoperability concerns, particularly when such analyses are performed in an automated manner. As a starting point, we plan to explore the adequacy of the Universal Variability Language (UVL) [169], which aims to unify diverse variability modeling approaches across domains. Although UVL provides

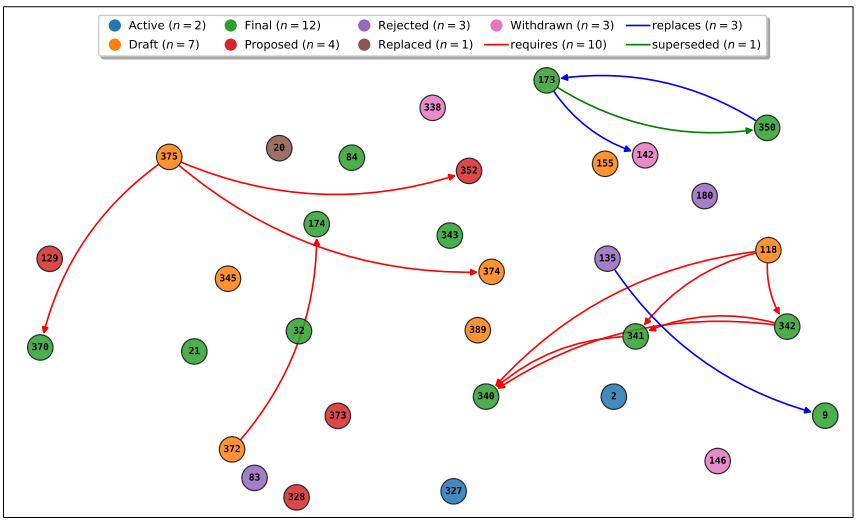
a flexible foundation, it may require extensions to support the specific constructs and metadata associated with IPs in CDV ecosystems.

Moreover, given the evolutionary and dynamic character of CDV, we aim to lift temporal evolution as a primary modeling concern. Unlike conventional variability modeling in software product-line engineering, which focuses primarily on variability in space during domain analysis, we propose a holistic modeling approach that superimposes variability in both space and time [47]. Specifically, we envision a formal definition of IP life cycles, including valid states and state transitions, as well as mechanisms to capture and later on reason about actual transitions as they occur in a CDV ecosystem. The conceptual research challenge is to design a modeling notation that supports the superimposition of variability in time and space, and serves as a basis for automated spatio-temporal analysis. As a starting point for addressing this challenge, we consider existing advanced modeling frameworks such as Hyper Feature Models (HFMs) [170] and Dynamic Feature Transition Systems (DFTS) [171]. HFMs extend traditional feature models to capture evolution over time, albeit currently limited to simple versioning scenarios. Still, this direction is promising for reflecting the multi-dimensional nature of CDV. Similarly, DFTSs, though designed for dynamic software product lines (DSPLs) with runtime reconfiguration, offer a context-aware transition model that could potentially be adapted to capture CDV dynamics.

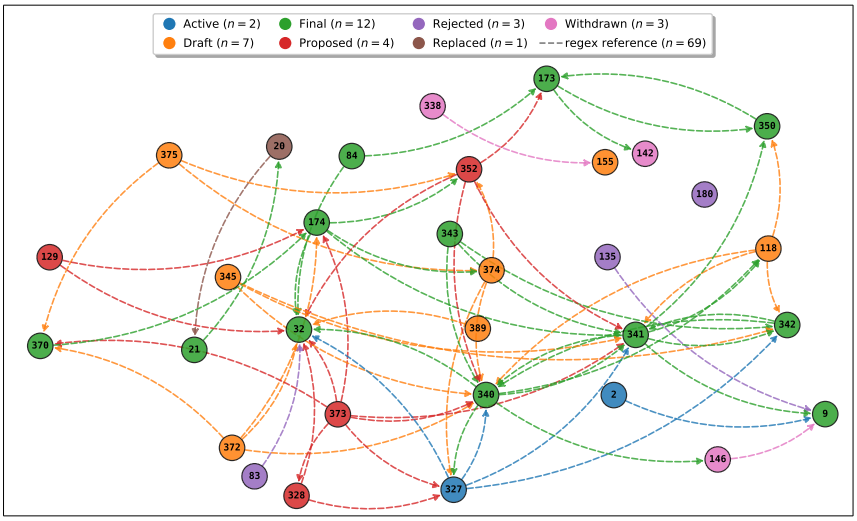
On the analysis side, representing CDV evolution based on transition systems enables the application of temporal logic to formally specify and verify evolution constraints. For instance, a constraint such as "If IP A requires IP B, then B must be active before or at the same time as A becomes active" can be expressed using linear temporal logic (LTL) [172] in combination with classical variability constraints (e.g., propositional logic). As for advanced analyses of the proposal spectrum evolution, the idea of semantic feature model differencing [173] may be adapted to the differential analysis of CDV model snapshots. The corresponding empirical research task is to evaluate the applicability and effectiveness of such analyses in real-world CDV ecosystems. On the methodological side, the task involves developing intelligent approaches to sustainably streamline efforts in formal specification and differential analysis of derivatives, leveraging their untapped potential for automation.

Variability model mining and IP consistency checks: As a preparatory step towards variability model mining, we have started to extract IP relationship graphs for the Bitcoin ecosystem. The nodes of these graphs represent individual BIPs, while the edges denote different kinds of inter-BIP relationships. Two such graphs are shown in Figure 6. The nodes are colored according to the BIPs' respective status[5] as of July 5, 2025. While both graphs represent the same subset of BIPs, the interelationships have been extracted using different methods. The edges in Figure 6a represent *explicitly* declared references to other BIPs, extracted from the respective fields (i.e., `Requires`, `Replaces`, and `Superseded-By`) declared by a BIP's preamble (cf. Figure 1). On the contrary, for the extraction results shown in Figure 6b, we sought to capture *implicit* references by scanning entire BIP documents for mentions of other BIPs using regular expressions. As these references cannot be classified into any categories using this

[5] See specification of the BIP status field in [25].



(a) Explicit BIP references from BIP2 [25] preamble.



(b) Implicit BIP references extracted through regex-search in document.

Figure 6: IP relationship graph for BIPs.

simple method of regex extraction, we color-code the edges with their source node's color.

While IP relationships will only cover one of the many aspects of a holistic model of CDV in the proposal spectrum, the graphs shown in Figure 6 already reveal first interesting insights. For example, looking at Figure 6a, one can see that, e.g., BIP340 seems to be important as it is required by 3 other BIPs. Also worth noting is that

there has been a replacement attempt for BIP9, which still resides in 'final' state, by BIP135 which itself has been rejected by the community. A similar situation can be observed with BIP173 and BIP350, where the latter replaces the former despite both residing in 'final' state. Derivatives (C3) may implement both IPs, or only one of them, or neither, invoking particular challenges for interoperability (C4).

Even more interestingly, however, our simple analysis reveals a substantial gap between explicitly declared dependencies and those being extracted by our regex-based full text analysis of IP documents. Although we acknowledge that the regex-based extraction of implicit references is a rather naive approach that may lack precision, it reveals a substantial amount of additional interdependencies that are not captured by the explicit references declared in the BIP's preambles. We manually checked a subset of the reported relationships and can confirm a considerable portion of true positives. For example, BIP20 and BIP21 referencing each other revealed a clearly missing preamble reference as the former *"has been replaced by BIP21"* [174]. Further one can see that there exist particularly important BIPs which receive 5 or more incoming edges. Among them BIP32, hierarchical deterministic wallets and de-facto standard by now (cf. Section 1), as well as BIP340 and 341, both pivotal for the latest transaction type known as Taproot [175]. Notably, BIP373, which currently holds *proposed* status, exhibits seven outgoing edges. Of these, five point to *final* BIPs, one to the *active* BIP327, and one to the *proposed* BIP328. This indicates that BIP373 is a highly integrating proposal, as it builds upon multiple established specifications. Verifying this observation in the BIP document confirms this role: it *"proposes additional fields for BIP174 PSBTv0 and BIP370 PSBTv2 that allow for BIP327 MuSig2 Multi-Signature data to be included in a PSBT of any version"*. [176]

On the one hand, our preliminary results encourage a more thorough investigation of effective techniques for IP relationship extraction. On the other hand, having a portfolio of different extraction techniques and comparing the IP relationship graphs seems to be an effective method for automatically identifying inconsistencies in IP specifications and basic anomalies in the proposal spectrum. Given Figure 6 is only showing a small and curated subset of the entire BIP catalog, we anticipate that an in-depth analysis at larger scale would reveal a lot of additional anomalies of different impact levels. In the long run, we envision that such automated anomaly detection techniques could become part of the standard tooling arsenal for ecosystem developers.

7.2.2 Starting Points for RG2 and RG3

The major task for realizing RG2 revolves around supporting IP traceability from the proposal to the derivative spectrum. We envision retroactive IP location techniques [177], as the dynamic nature of these ecosystems often hinders proactive IP tracing. Since we cannot assume the derivatives being created through traditional clone-and-own [12, 13], yielding sets of variants exposing only minor divergences, we may hardly adopt set-based techniques such as Ecco [178] for this task. However, it might be promising to evaluate the performance of feature location techniques being capable of working with single variants only [179]. Moreover, since derivatives most

likely integrate reference libraries such as cryptographic primitives, extracting Software Bills of Materials (SBOMs) [180] for derivatives may inform the identification of their implemented IPs.

The most challenging part of RG3 is to support the detection of unanticipated IP interactions impairing interoperability. While pushing the boundaries from intra-derivative to inter-derivative interaction testing goes beyond software quality issues addressed by software product-line testing [181], it exposes similar challenges. As testing all the mutual IP interactions of implementation derivatives is infeasible, we strive for novel sampling methods that enable spotting the most harmful interactions effectively. To that end, we aim to lift existing combinatorial interaction testing strategies [182] to CDV models. This allows us to systematically explore the sample space induced by different strategies and eventually making informed decisions in balancing efficiency and effectiveness.

Despite our specific focus on CDV, we acknowledge the substantial body of research dedicated to interoperability and the evolution of standards across various domains. This research primarily aims to provide comprehensive overviews of existing standards and enhance interoperability between heterogeneous systems through standardization [183]. Besides the aforementioned IoT domain [146–152], this includes also supply chain management [184], eHealth systems [185], PDF viewers [186], or cloud computing [187, 188]. While these efforts typically target systems with distinct business purposes that must interoperate, they offer valuable insights that may complement our perspective on CDV ecosystems. We see potential for mutual benefit through methodological exchange and conceptual alignment between these research directions.

7.2.3 Research Methods and Study Subjects

Aligned with our technical goals, we adopt a design science approach, implementing conceptual solutions as prototypes and prioritizing internal over external validity in evaluation [189]. We will first focus on the Bitcoin ecosystem for three reasons: (1) its large community and high degree of CDV, (2) the abundance of high-quality, openly available data, and (3) its long history, allowing for retrospective study and simulation of its dynamics. Next, we increase the external validity of our results by studying other ecosystems with similar characteristics. In parallel, we will conduct qualitative research through surveys and interviews with actors of CDV ecosystems, for further validation and potential refinement of our problem analysis. Furthermore, we will explore the impact of our research on ecosystems that are closely related to CDV.

8 Conclusion

Community-Driven Variability (CDV) represents an emerging form of distributed software variability that is unexplored in current literature and transcends traditional variability-intensive systems: Instead of a monolithic stakeholder centrally driving and controlling all aspects of variability, a distributed community iteratively and independently shapes the ecosystem by agreeing on a set of necessary interfaces, which forms a constantly evolving implicit standard that strives for interoperability. This vibrant field offers a number of relevant challenges, which become increasingly complex as the communities grow and the ecosystems evolve.

In this paper, we provided a comprehensive definition of the five constituting characteristics of CDV and applied them in the evaluation of 14 ecosystems. From the perspective of automated software engineering, this conceptual work serves as a foundational form of requirements analysis, laying the groundwork for both the automation techniques we envision and the broader line of research that will build upon it. Our research vision, composed of three goals, leverages feature-oriented modeling and analysis concepts as a promising starting point for systematically addressing CDV-specific challenges. Crucially, our approach refrains from imposing conventional product-line processes, thereby fostering methodological synergies and enabling mutual impact across community-driven and traditional paradigms. Instead, we envision that techniques from established variability paradigms can be fruitfully transferred to CDV contexts – provided that the underlying structural and organizational characteristics are compatible (cf. Table 1). Conversely, insights gained from CDV may inform and enrich those paradigms. We envision such reciprocal methodological enrichment to advance both research and practice across variability-intensive domains.

Our preliminary investigation into relationships among Bitcoin Improvement Proposals (BIPs) already uncovered inconsistencies and coordination gaps (cf. Figure 6), illustrating the pressing need for automated, systematic, and tool-supported approaches to capture and manage the inherent complexity in CDV ecosystems. We thus invite the software engineering community to engage with the challenges and opportunities posed by CDV. Advancing our understanding, modeling capabilities, and support mechanisms for CDV ecosystems will improve current practice and contribute to a broader rethinking of variability in open, collaborative software development contexts.

References

[1] Parnas, D.L.: On the design and development of program families. IEEE Trans. on Software Engineering (TSE) (1), 1–9 (1976)

[2] Pohl, K., Böckle, G., Linden, F.J.: Software Product Line Engineering: Foundations, Principles and Techniques. Springer (2005)

[3] Apel, S., Batory, D.S., Kästner, C., Saake, G.: Feature-Oriented Software Product Lines - Concepts and Implementation. Springer (2013)

[4] Batory, D.S.: Feature models, grammars, and propositional formulas. In: Proc. Int'l Systems and Software Product Line Conf. (SPLC). Lecture Notes in Computer Science (LNCS), vol. 3714, pp. 7–20. Springer (2005)

[5] Czarnecki, K., Eisenecker, U.W.: Generative Programming - Methods, Tools and Applications. Addison-Wesley, Boston, USA (2000)

[6] Svahnberg, M., Gurp, J., Bosch, J.: A taxonomy of variability realization techniques. Software: Practice and Experience **35**(8), 705–754 (2005)

[7] Eggert, M., Günther, K., Maletschek, J., Maxiniuc, A., Mann-Wahrenberg, A.: In three steps to software product lines: a practical example from the automotive industry. In: Proceedings of the 26th ACM International Systems and Software Product Line Conference - Volume A. SPLC'22, pp. 170–177. Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3546932.3547003

[8] Habli, I., Kelly, T., Hopkins, I.: Challenges of establishing a software product line for an aerospace engine monitoring system. In: 11th International Software Product Line Conference (SPLC 2007), pp. 193–202 (2007). https://doi.org/10.1109/SPLINE.2007.37

[9] Abbas, M., Jongeling, R., Lindskog, C., Enoiu, E.P., Saadatmand, M., Sundmark, D.: Product line adoption in industry: an experience report from the railway domain. In: Proceedings of the 24th ACM Conference on Systems and Software Product Line: Volume A - Volume A. SPLC '20. Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3382025.3414953

[10] Myllärniemi, V., Savolainen, J., Männistö, T.: Performance variability in software product lines: a case study in the telecommunication domain. In: Proceedings of the 17th International Software Product Line Conference. SPLC '13, pp. 32–41. Association for Computing Machinery, New York, NY, USA (2013). https://doi.org/10.1145/2491627.2491631

[11] She, S., Lotufo, R., Berger, T., Wasowski, A., Czarnecki, K.: The variability model of the linux kernel. In: Proc. Int'l Workshop on Variability Modelling of Software-Intensive Systems (VaMoS), vol. 37, pp. 45–51 (2010)

[12] Rubin, J., Czarnecki, K., Chechik, M.: Managing Cloned Variants: A Framework and Experience. In: Proc. Int'l Systems and Software Product Line Conf. (SPLC), pp. 101–110. ACM, New York, NY, USA (2013)

[13] Kehrer, T., Thüm, T., Schultheiß, A., Bittner, P.M.: Bridging the Gap Between Clone-and-Own and Software Product Lines. In: Proc. Int'l Conf. on Software Engineering (ICSE), pp. 21–25. IEEE, Piscataway, NJ, USA (2021)

[14] Schmorleiz, T., Lämmel, R.: Similarity Management via History Annotation. In: Proc. Seminar on Advanced Techniques and Tools for Software Evolution (SATToSE), pp. 45–48. Dipartimento di Informatica Università degli Studi dell'Aquila, L'Aquila, Italy (2014)

[15] Wang, L., Zheng, Z., Wu, X., Sang, B., Zhang, J., Tao, X.: Fork entropy: Assessing the diversity of open source software projects' forks. In: 38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023, Luxembourg, September 11-15, 2023, pp. 204–216 (2023). https://doi.org/10.1109/ASE56229.2023.00168

[16] Krüger, J., Mukelabai, M., Gu, W., Shen, H., Hebig, R., Berger, T.: Where Is My Feature and What Is It About? A Case Study on Recovering Feature Facets. J. Systems and Software (JSS) **152**, 239–253 (2019)

[17] Linsbauer, L., Lopez-Herrejon, R.E., Egyed, A.: Variability Extraction and Modeling for Product Variants. Software and System Modeling (SoSyM) **16**(4), 1179–1199 (2017)

[18] Klatt, B., Küster, M., Krogmann, K.: A Graph-Based Analysis Concept to Derive a Variation Point Design From Product Copies. In: Proc. Int'l Workshop on Reverse Variability Engineering (REVE), pp. 1–8 (2013)

[19] Kästner, C., Dreiling, A., Ostermann, K.: Variability Mining: Consistent Semiautomatic Detection of Product-Line Features. IEEE Trans. on Software Engineering (TSE) **40**(1), 67–82 (2014)

[20] Martinez, J., Ziadi, T., Bissyandé, T.F., Klein, J., Le Traon, Y.: Bottom-Up Adoption of Software Product Lines: A Generic and Extensible Approach. In: Proc. Int'l Systems and Software Product Line Conf. (SPLC), pp. 101–110. ACM, New York, NY, USA (2015)

[21] Fenske, W., Meinicke, J., Schulze, S., Schulze, S., Saake, G.: Variant-Preserving Refactorings for Migrating Cloned Products to a Product Line. In: Proc. Int'l Conf. on Software Analysis, Evolution and Reengineering (SANER), pp. 316–326. IEEE, Piscataway, NJ, USA (2017)

[22] Rosu, C., Togan, M.: A modern paradigm for effective software development: Feature toggle systems. In: 15th International Conference on Electronics, Computers and Artificial Intelligence, ECAI 2023, June 29-30, pp. 1–6. IEEE, Bucharest (2023). https://doi.org/10.1109/ECAI58194.2023.10193936

[23] Mahdavi-Hezaveh, R., Dremann, J., Williams, L.A.: Software development with feature toggles: practices used by practitioners. Empir. Softw. Eng. **26**(1), 1 (2021) https://doi.org/10.1007/S10664-020-09901-Z

[24] Krüger, J., Berger, T.: An Empirical Analysis of the Costs of Clone- and Platform-Oriented Software Reuse. In: Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE), pp. 432–444. ACM (2020)

[25] Dashjr, L.: BIP2: BIP Process, Revised. https://bips.dev/2 Accessed 2025-01-16

[26] Nakamoto, S.: Bitcoin: A Peer-to-Peer Electronic Cash System (2008)

[27] Bitcoin Improvement Proposals (BIPs). https://github.com/bitcoin/bips Accessed 2025-01-16

[28] Benet, J.: IPFS - Content Addressed, Versioned, P2P File System. arXiv (2014)

[29] InterPlanetary Improvement Proposals (IPIPs). https://specs.ipfs.tech/ipips Accessed 2025-01-16

[30] Dingledine, R., Mathewson, N., Syverson, P.F.: Tor: The second-generation onion router. In: Blaze, M. (ed.) Proceedings of the 13th USENIX Security Symposium, August 9-13, 2004, San Diego, CA, USA, pp. 303–320. USENIX, Berkeley, CA (2004)

[31] Tor Design Proposals. https://spec.torproject.org/proposals/index.html Accessed 2025-01-16

[32] nostr-protocol/nostr. nostr-protocol. https://github.com/nostr-protocol/nostr Accessed 2024-12-16

[33] Nostr Implementation Possibilities (NIPs). https://github.com/nostr-protocol/nips Accessed 2025-01-16

[34] Erhardt, M.: Re: [Bitcoindev] Time for an Update to BIP2? Accessed: 2025-01-16 (2024). https://mailing-list.bitcoindevs.xyz/bitcoindev/82a37738-a17b-4a8c-9651-9e241118a363@murch.one

[35] Amichateur: [Awareness/Proposal] The Multitude of Different "Derivation Paths" in BIP32 Bitcoin Wallets Causes Incompatibility All Around When It Comes to Wallet Seed Restore Operation for Non-Tech-Savy Users. r/Bitcoin. https://www.reddit.com/r/Bitcoin/comments/qeu3j7/awarenessproposal_the_multitude_of_different/ Accessed 2025-01-16

[36] Wuille, P.: BIP32: Hierarchical Deterministic Wallets. https://bips.dev/32 Accessed 2025-01-16

[37] Questions Tagged [bip32-hd-wallets]. Bitcoin Stack Exchange. https://bitcoin.stackexchange.com/questions/tagged/bip32-hd-wallets Accessed 2025-01-16

[38] Wallets Recovery - Bitcoin Wallet Seeds Recovery Guide. https://walletsrecovery.org/ Accessed 2025-07-20

[39] Bitcoin Optech: Compatibility Matrix. https://bitcoinops.org/en/compatibility Accessed 2025-01-16

[40] Choose Your Wallet. Bitcoin.org. https://bitcoin.org/en/choose-your-wallet Accessed 2025-01-16

[41] Know Your Wallet Like You Built It. WalletScrutiny. https://walletscrutiny.com Accessed 2025-01-16

[42] Software Wallets: Comparing 25 Bitcoin Software Wallets Feature by Feature. The Bitcoin Hole. https://thebitcoinhole.com/software-wallets Accessed 2025-07-20

[43] Bögli, R., Boll, A., Schultheiß, A., Kehrer, T.: Beyond software families: Community-driven variability. In: Montecchi, L., Li, J., Poshyvanyk, D., Zhang, D. (eds.) 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, pp. 571–575. ACM (2025). https://doi.org/10.1145/3696630.3728501

[44] Mistrík, I., Galster, M., Maxim, B.R. (eds.): Software Engineering for Variability Intensive Systems - Foundations and Applications. Auerbach Publications / Taylor & Francis, New York (2019). https://doi.org/10.1201/9780429022067

[45] Dijkstra, E.W.: The humble programmer. Communications of the ACM **15**(10), 859–866 (1972)

[46] Lehman, M.M.: Laws of software evolution revisited. In: European Workshop on Software Process Technology, pp. 108–124 (1996). Springer

[47] Ananieva, S., Greiner, S., Kehrer, T., Krüger, J., Kühn, T., Linsbauer, L., Grüner, S., Koziolek, A., Lönn, H., Ramesh, S., *et al.*: A conceptual model for unifying variability in space and time: Rationale, validation, and illustrative applications. Empirical Software Engineering **27**(5), 101 (2022)

[48] Svahnberg, M., Van Gurp, J., Bosch, J.: A taxonomy of variability realization techniques. Software: Practice and experience **35**(8), 705–754 (2005)

[49] Stănciulescu, Ş., Schulze, S., Wąsowski, A.: Forked and integrated variants in an open-source firmware project. In: 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 151–160 (2015). IEEE

[50] Rowe, D., Leaney, J., Lowe, D.: Defining Systems Evolvability - A Taxonomy of Change. In: Proc. Int'l Conf. on Engineering of Computer-Based Systems (ECBS), pp. 45–52. IEEE Computer Society. https://doi.org/10.1109/ECBS.1998.10027

[51] Breivold, H.P., Crnkovic, I., Eriksson, P.: Evaluating Software Evolvability. Software Engineering Research and Practice in Sweden **96** (2007)

[52] Zave, P.: An experiment in feature engineering. In: McIver, A., Morgan, C. (eds.) Programming Methodology, pp. 353–377. Springer, New York (2003). https://doi.org/10.1007/978-0-387-21798-7_17

[53] Batory, D., Sarvela, J.N., Rauschmayer, A.: Scaling step-wise refinement. IEEE Transactions on Software Engineering **30**(6), 355–371 (2004)

[54] Thüm, T., Apel, S., Kästner, C., Schaefer, I., Saake, G.: A classification and survey of analysis strategies for software product lines. ACM Computing Surveys (CSUR) **47**(1), 1–45 (2014)

[55] Berger, T., Chechik, M., Kehrer, T., Wimmer, M.: Software evolution in time and space: Unifying version and variability management (dagstuhl seminar 19191). Dagstuhl Reports **9**(5), 1–30 (2019)

[56] Xiao, Y., Zhang, N., Lou, W., Hou, Y.T.: A Survey of Distributed Consensus Protocols for Blockchain Networks. IEEE Commun. Surv. Tutorials **22**(2), 1432–1465 (2020) https://doi.org/10.1109/COMST.2020.2969706

[57] craigraw: Sparrow Bitcoin Wallet. https://sparrowwallet.com/features Accessed 2025-01-16

[58] Palatinus, M., Rusnak, P., Aaron, V., Bowe, S.: BIP39: Mnemonic Code for Generating Deterministic Keys. https://bips.dev/39 Accessed 2025-01-16

[59] Electrum Bitcoin Wallet. https://electrum.org Accessed 2025-01-16

[60] SatoshiLabs: SatoshiLabs Improvement Proposals (SLIPs). https://github.com/satoshilabs/slips Accessed 2025-01-16

[61] scgbckbone: Commit 829afcc "change BIP39 Status to Final". bitcoin/bips on GitHub. https://github.com/bitcoin/bips/commit/829afccd1ae26403f8c3583d7347b04aeb54c2ca Accessed 2025-07-19

[62] Electrum Seed Version System. https://electrum.readthedocs.io/en/latest/seedphrase.html Accessed 2025-01-16

[63] Poon, J., Dryja, T.: The Bitcoin Lightning Network: Scalable Off-Chain Instant Payments (2016)

[64] Basis of Lightning Technology (BOLT). https://github.com/lightning/bolts Accessed 2025-01-16

[65] Bitcoin Lightning Improvement Proposals (bLIPs). https://github.com/lightning/blips Accessed 2025-01-16

[66] Buterin, V., *et al.*: A next-generation smart contract and decentralized application platform. white paper **3**(37), 2–1 (2014)

[67] Ethereum Improvement Proposals (EIPs). https://eips.ethereum.org Accessed 2025-01-16

[68] Torvalds, L., et al.: Linux Kernel Source Tree. Website: https://github.com/torvalds/linux. Accessed: 2025-07-12

[69] Torvalds, L., et al.: The Linux Kernel Archives. Website: https://www.kernel.org/linux.html. Accessed: 2025-07-12

[70] AISBL, E.F.: Eclipse. Website: https://www.eclipse.org/home/whatis/. Accessed: 2025-07-12

[71] Andersen, E., Vlasenko, D., et al.: BusyBox: The Swiss Army Knife of Embedded Linux. Website: https://busybox.net/about.html. Accessed: 2025-07-12

[72] Aporius, D.: Apo-Games. Website: https://www.apo-games.de/. Accessed: 2025-07-12

[73] Lahteine, S., et al.: Marlin Firmware. Website: https://marlinfw.org/docs/basics/introduction.html. Accessed: 2025-07-12

[74] Soares, S., Borba, P., Laureano, E.: Distribution and Persistence as Aspects. Software: Practice and Experience **36**(7), 711–759 (2006) https://doi.org/10.1002/SPE.715

[75] Documentation. Home Assistant. https://www.home-assistant.io/docs Accessed 2025-01-16

[76] Python. https://python.org Accessed 2025-01-16

[77] Python Enhancement Proposals (PEPs). https://peps.python.org Accessed 2025-01-16

[78] lnurl/luds. https://github.com/lnurl/luds Accessed 2025-07-17

[79] lightningnetwork/lnd. https://github.com/lightningnetwork/lnd Accessed 2025-07-15

[80] ACINQ/Eclair. https://github.com/ACINQ/eclair Accessed 2025-07-15

[81] ElementsProject/lightning. https://github.com/ElementsProject/lightning Accessed 2025-07-15

[82] BOLT9: Assigned Feature Flags. https://github.com/lightning/bolts/blob/master/09-features.md Accessed 2025-01-16

[83] van der Wijden, M., Lange, F., Rong, G.: EIP-4938: Eth/67 - Removal of GetNodeData. https://eips.ethereum.org/EIPS/eip-4938 Accessed 2025-07-16

[84] DuPont, Q.: Experiments in algorithmic governance: A history and ethnography of "the dao," a failed decentralized autonomous organization. In: Bitcoin and Beyond, pp. 157–177. Routledge, New York (2017)

[85] ethereum/go-ethereum. https://github.com/ethereum/go-ethereum Accessed 2025-07-16

[86] MetaMask/metamask-extension. https://github.com/MetaMask/metamask-extension Accessed 2025-07-16

[87] ProjectOpenSea/Seaport. https://github.com/ProjectOpenSea/seaport Accessed 2025-07-16

[88] ISO/IEC 7498-1:1994, Information Technology - Open Systems Interconnection - Basic Reference Model: The Basic Model. https://www.iso.org/standard/20269.html

[89] Argentieri, M.: NIP46 Nostr Remote Signing. https://github.com/nostr-protocol/nips/blob/master/46.md Accessed 2025-09-02

[90] Miller, P., Staab, J.: NIP44 Encrypted Payloads (Versioned). https://github.com/nostr-protocol/nips/blob/master/44.md Accessed 2025-09-02

[91] Castillo, M.: Meet @Fiatjaf, The Mysterious Nostr Creator Who Has Lured 18 Million Users And $5 Million From Jack Dorsey. https://www.forbes.com/sites/digital-assets/2023/05/30/bitcoin-social-network-nostr-creator-fiatjaf-/ Accessed 2025-07-16

[92] sandwichfarm/nostr-watch: A NIP-66 Nostr Client for Browsing Nostr Relays. https://github.com/sandwichfarm/nostr-watch Accessed 2025-07-14

[93] aljazceru/Awesome-Nostr. https://github.com/aljazceru/awesome-nostr Accessed 2025-07-14

[94] Explore Nostr Apps. https://nostrapps.com/ Accessed 2025-07-14

[95] Tor Browser. GitLab. https://gitlab.torproject.org/tpo/applications/tor-browser Accessed 2025-07-17

[96] Perry, M.: 324-Rtt-Congestion-Control (2020). https://spec.torproject.org/proposals/324-rtt-congestion-control.html Accessed 2025-07-17

[97] Perry, M.: 291-Two-Guard-Nodes (2018). https://spec.torproject.org/proposals/291-two-guard-nodes.html Accessed 2025-07-17

[98] The Open Database Of The Corporate World: THE TOR PROJECT, INC. https://opencorporates.com/companies/us_ma/208096820 Accessed 2025-07-17

[99] The Tor Project. GitLab. https://gitlab.torproject.org/tpo Accessed 2025-07-18

[100] Mathewson, N.: 264-Subprotocol-Versions - Tor Design Proposals (2016). https://spec.torproject.org/proposals/264-subprotocol-versions.html Accessed 2025-07-17

[101] Mathewson, N.: 346-Protovers-Again - Tor Design Proposals (2023). https://spec.torproject.org/proposals/346-protovers-again.html Accessed 2025-07-17

[102] Doan, T.V., Psaras, Y., Ott, J., Bajpai, V.: Toward decentralized cloud storage with ipfs: Opportunities, challenges, and future considerations. IEEE Internet Computing **26**(6), 7–15 (2022) https://doi.org/10.1109/MIC.2022.3209804

[103] Protocol Labs. https://protocol.ai/ Accessed 2025-07-17

[104] ipfs/Kubo. IPFS Project. https://github.com/ipfs/kubo Accessed 2025-07-17

[105] ipfs/Helia. IPFS Project. https://github.com/ipfs/helia Accessed 2025-07-17

[106] ipfs/awesome-ipfs: Community List of Awesome Projects, Apps, Tools, Pinning Services and More Related To IPFS. https://github.com/ipfs/awesome-ipfs/tree/main Accessed 2025-07-17

[107] Krueger, C.W.: Easing the Transition to Software Mass Customization. In: Proc. Int'l Workshop on Software Product-Family Engineering (PFE), pp. 282–293. Springer (2002)

[108] Pohl, K., Böckle, G., Linden, F.J.: Software Product Line Engineering: Foundations, Principles and Techniques. Springer (2005)

[109] Apel, S., Batory, D., Kästner, C., Saake, G.: Feature-Oriented Software Product Lines. Springer (2013)

[110] Czarnecki, K., Eisenecker, U.: Generative Programming: Methods, Tools, and Applications. ACM/Addison-Wesley (2000)

[111] Batory, D.: Feature Models, Grammars, and Propositional Formulas. In: Proc. Int'l Systems and Software Product Line Conf. (SPLC), pp. 7–20. Springer (2005)

[112] Kconfig Language. The Linux Kernel documentation. https://www.kernel.org/doc/html/next/kbuild/kconfig-language.html Accessed 2025-07-21

[113] Sincero, J., Schirmeier, H., Schröder-Preikschat, W., Spinczyk, O.: Is the Linux Kernel a Software Product Line? In: Proc. Int'l Workshop on Open Source Software and Product Lines (OSSPL), pp. 9–12. IEEE (2007)

[114] Abal, I., Melo, J., Stănciulescu, S., Brabrand, C., Ribeiro, M., Wąsowski, A.: Variability Bugs in Highly Configurable Systems: A Qualitative Analysis. Trans. on Software Engineering and Methodology (TOSEM) **26**(3), 10–11034 (2018)

[115] Mortara, J., Collet, P.: Capturing the Diversity of Analyses on the Linux Kernel Variability. In: Proc. Int'l Systems and Software Product Line Conf. (SPLC), pp. 160–171. ACM (2021)

[116] Franz, P., Berger, T., Fayaz, I., Nadi, S., Groshev, E.: ConfigFix: Interactive Configuration Conflict Resolution for the Linux Kernel. In: Proc. Int'l Conf. on Software Engineering: Software Engineering in Practice (ICSE-SEIP), pp. 91–100. IEEE (2021)

[117] Kuiter, E., Sundermann, C., Thüm, T., Hess, T., Krieter, S., Saake, G.: How Configurable is the Linux Kernel? Analyzing Two Decades of Feature-Model History. Trans. on Software Engineering and Methodology (TOSEM) (2025)

[118] Eclipse Foundation. Eclipse Foundation. https://www.eclipse.org/home/ Accessed 2025-07-03

[119] Gurp, J., Prehofer, C., Bosch, J.: Comparing practices for reuse in integration-oriented software product lines and large open source software projects. Software: Practice and Experience **40**(4), 285–312 (2010)

[120] Pleuss, A., Botterweck, G., Dhungana, D., Polzer, A., Kowalewski, S.: Model-driven support for product line evolution on feature level. J. Systems and Software (JSS) **85**(10), 2261–2274 (2012)

[121] Ahmed, F., Capretz, L.F., Babar, M.A.: A Model of Open Source Software-Based Product Line Development. In: Proc. on Computer Software and Applications Conf. (COMPSAC), pp. 1215–1220. IEEE (2008)

[122] Heider, W., Rabiser, R., Grünbacher, P.: Facilitating the evolution of products in product line engineering by capturing and replaying configuration decisions. Int'l J. Software Tools for Technology Transfer (STTT) **14**(5), 613–630 (2012)

[123] Cervantes, H., Charleston-Villalobos, S.: Using a lightweight workflow engine in a plugin-based product line architecture. In: Proc. of the Int'l Symposium on Component-Based Software Engineering (CBSE). Lecture Notes in Computer Science (LNCS), vol. 4063, pp. 198–205. Springer (2006)

[124] Schultheiß, A., Bittner, P.M., Thüm, T., Kehrer, T.: Quantifying the Potential to Automate the Synchronization of Variants in Clone-and-Own. In: Proc. Int'l Conf. on Software Maintenance and Evolution (ICSME), pp. 269–280. IEEE, Piscataway, NJ, USA (2022)

[125] Wang, A., Feng, N., Chechik, M.: Code-Level Functional Equivalence Checking of Annotative Software Product Lines. In: Proc. Int'l Systems and Software Product Line Conf. (SPLC), pp. 64–75. ACM (2023)

[126] Pett, T., Krieter, S., Runge, T., Thüm, T., Lochau, M., Schaefer, I.: Stability of Product-Line Sampling in Continuous Integration. In: Proc. Int'l Working Conf. on Variability Modelling of Software-Intensive Systems (VaMoS). ACM (2021)

[127] Friesel, B., Müller, M., Ferraz, M.F., Spinczyk, O.: On the Relation of Variability Modeling Languages and Non-Functional Properties. In: Proc. Int'l Systems and Software Product Line Conf. (SPLC), pp. 140–144. ACM (2022)

[128] Pett, T., Krieter, S., Thüm, T., Schaefer, I.: MulTi-Wise Sampling: Trading Uniform T-Wise Feature Interaction Coverage for Smaller Samples. In: Proc. Int'l Systems and Software Product Line Conf. (SPLC), pp. 47–53. ACM (2024)

[129] Bombarda, A., Gargantini, A.: On the Use of Multi-valued Decision Diagrams to Count Valid Configurations of Feature Models. In: Proc. Int'l Systems and Software Product Line Conf. (SPLC), pp. 96–106. ACM (2024)

[130] Stănciulescu, S., Schulze, S., Wąsowski, A.: Forked and Integrated Variants in an Open-Source Firmware Project. In: Proc. Int'l Conf. on Software Maintenance and Evolution (ICSME), pp. 151–160. IEEE (2015)

[131] Krüger, J., Fenske, W., Thüm, T., Aporius, D., Saake, G., Leich, T.: Apo-Games: A Case Study for Reverse Engineering Variability From Cloned Java Variants. In: Proc. Int'l Systems and Software Product Line Conf. (SPLC), pp. 251–256. ACM (2018)

[132] Kim, T., Lee, J., Kang, S.: Cloned Code Clustering for the Software Product Line Engineering Approach to Developing a Family of Products. In: Proc. on Computer Software and Applications Conf. (COMPSAC), pp. 1350–1355. IEEE (2024)

[133] Marchezan, L., Assunção, W.K.G., Michelon, G.K., Herac, E., Egyed, A.: Code Smell Analysis in Cloned Java Variants: The Apo-Games Case Study. In: Proc. Int'l Systems and Software Product Line Conf. (SPLC), pp. 250–254. ACM (2022)

[134] Debbiche, J., Lignell, O., Krüger, J., Berger, T.: Migrating Java-Based Apo-Games into a Composition-Based Software Product Line. In: Proc. Int'l Systems and Software Product Line Conf. (SPLC), pp. 98–102. ACM (2019)

[135] Åkesson, J., Nilsson, S., Krüger, J., Berger, T.: Migrating the Android Apo-Games Into an Annotation-Based Software Product Line. In: Proc. Int'l Systems and Software Product Line Conf. (SPLC), pp. 103–107. ACM (2019)

[136] Schultheiß, A., Bittner, P.M., Boll, A., Grunske, L., Thüm, T., Kehrer, T.: RaQuN: A Generic and Scalable N-Way Model Matching Algorithm. Software and System Modeling (SoSyM) **22**, 1495–1517 (2023)

[137] Zhou, S., Vasilescu, B., Kästner, C.: What the Fork: A Study of Inefficient and Efficient Forking Practices in Social Coding. In: Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE), pp. 350–361. ACM (2019)

[138] Krüger, J., Gu, W., Shen, H., Mukelabai, M., Hebig, R., Berger, T.: Towards a Better Understanding of Software Features and Their Characteristics: A Case Study of Marlin. In: Proc. Int'l Workshop on Variability Modelling of Software-Intensive Systems (VaMoS), pp. 105–112. ACM (2018)

[139] Stanciulescu, S., Rabiser, D., Seidl, C.: A Technology-Neutral Role-Based Collaboration Model for Software Ecosystems. In: Proc. Int'l Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA). Lecture Notes in Computer Science (LNCS), vol. 9953, pp. 512–530 (2016)

[140] Linsbauer, L., Berger, T., Grünbacher, P.: A Classification of Variation Control Systems. In: Proc. Int'l Conf. on Generative Programming: Concepts & Experiences (GPCE), pp. 49–62. ACM (2017)

[141] Shatnawi, A., Seriai, A., Sahraoui, H.A.: Recovering Architectural Variability of a Family of Product Variants. In: Proc. Int'l Conf. on Software Reuse (ICSR). Lecture Notes in Computer Science (LNCS), vol. 8919, pp. 17–33. Springer (2015)

[142] Shatnawi, A., Seriai, A., Sahraoui, H.A.: Recovering Software Product Line Architecture of a Family of Object-Oriented Product Variants. J. Systems and Software (JSS) **131**, 325–346 (2017)

[143] Shatnawi, A., Seriai, A., Sahraoui, H.A., Ziadi, T., Seriai, A.: ReSIde: Reusable Service Identification from Software Families. J. Systems and Software (JSS) **170**, 110748 (2020)

[144] Lima, C., Machado, I., Galster, M., Flach G. Chavez, C.: Recovering Architectural Variability from Source Code. In: Proc. Brazilian Symposium on Software Engineering (SBES), pp. 808–817. ACM (2020)

[145] Greenwood, P., Bartolomei, T.T., Figueiredo, E., Dósea, M., Garcia, A.F., Cacho, N., Sant'Anna, C., Soares, S., Borba, P., Kulesza, U., Rashid, A.: On the Impact of Aspectual Decompositions on Design Stability: An Empirical Study. In: Proc. Europ. Conf. on Object-Oriented Programming (ECOOP). Lecture Notes in Computer Science (LNCS), vol. 4609, pp. 176–200. Springer (2007)

[146] Lee, E., Seo, Y., Oh, S., Kim, Y.: A survey on standards for interoperability and security in the internet of things **23**(2), 1020–1047 (2021)

[147] Noura, M., Atiquzzaman, M., Gaedke, M.: Interoperability in internet of things: Taxonomies and open challenges **24**(3), 796–809 (2019)

[148] Martino, B.D., Rak, M., Ficco, M., Esposito, A., Maisto, S.A., Nacchia, S.: Internet of things reference architectures, security and interoperability: A survey. Internet Things **1-2**, 99–112 (2018)

[149] Gyrard, A., Datta, S.K., Bonnet, C.: A survey and analysis of ontology-based software tools for semantic interoperability in iot and wot landscapes, pp. 86–91. IEEE, World Forum on Internet of Things (WF-IoT) (2018)

[150] Burzlaff, F., Wilken, N., Bartelt, C., Stuckenschmidt, H.: Semantic interoperability methods for smart service systems: A survey **69**(6), 4052–4066 (2022)

[151] Kambourakis, G., Kolias, C., Geneiatakis, D., Karopoulos, G., Makrakis, G.M., Kounelis, I.: A state-of-the-art review on the security of mainstream iot wireless PAN protocol stacks. Symmetry **12**(4), 579 (2020)

[152] Ganzha, M., Paprzycki, M., Pawlowski, W., Szmeja, P., Wasielewska, K.: Semantic interoperability in the internet of things: An overview from the inter-iot perspective. J. Netw. Comput. Appl. **81**, 111–124 (2017)

[153] Nabu Casa: About Us. Nabu Casa. https://www.nabucasa.com/about/ Accessed 2025-07-17

[154] Sørensen, J.: HACS 2.0 - The Best Way to Share Community-Made Projects Just Got Better. Home Assistant. https://www.home-assistant.io/blog/2024/08/21/hacs-the-best-way-to-share-community-made-projects/ Accessed 2025-07-17

[155] Home Assistant Community Store (HACS). https://hacs.xyz/ Accessed 2025-07-17

[156] marshalleq: Thinking of Dropping Home Assistant Due to Poor Integration Connectivity and Consistency. Home Assistant Community. https://community.home-assistant.io/t/thinking-of-dropping-home-assistant-due-to-poor-integration-connectivity-and-consistency/779978 Accessed 2025-07-17

[157] Python Package Index (PyPI). https://pypi.org/ Accessed 2025-07-18

[158] About the Python Software Foundation. Python.org. https://www.python.org/psf/about/ Accessed 2025-07-18

[159] PEP 13 – Python Language Governance. Python Enhancement Proposals (PEPs). https://peps.python.org/pep-0013/ Accessed 2025-07-18

[160] Micropython/Micropython. MicroPython. https://github.com/micropython/micropython Accessed 2025-07-18

[161] RustPython/RustPython. RustPython. https://github.com/RustPython/RustPython Accessed 2025-07-18

[162] MicroPython Differences from CPython. MicroPython Documentation. https://docs.micropython.org/en/latest/genrst/index.html Accessed 2025-07-18

[163] Bech32 Adoption. Bitcoin Wiki. https://en.bitcoin.it/wiki/Bech32_adoption

[164] Lopp, J.: Bitcoin Technical Resources. https://www.lopp.net/bitcoin-information/technical-resources.html

[165] Chan, W.K., Chin, J.-J., Goh, V.T.: Evolution of Bitcoin Addresses from Security Perspectives. In: ICITST, pp. 1–6 (2020)

[166] Bertrand, J.: The Hidden World of Bitcoin Invalid Blocks: Insights And Implications. D-Central. https://d-central.tech/the-hidden-world-of-bitcoin-invalid-blocks-insights-and-implications Accessed 2025-01-16

[167] Bier, J.: The Blocksize War: The Battle over Who Controls Bitcoin's Protocol Rules. Independently published, n.p. (2021)

[168] Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S.: Feature-oriented domain analysis (foda) feasibility study. Technical report (1990)

[169] Benavides, D., Sundermann, C., Feichtinger, K., Galindo, J.A., Rabiser, R., Thüm, T.: Uvl: Feature modelling with the universal variability language. Journal of systems and software **225**, 112326 (2025)

[170] Seidl, C., Schaefer, I., Aßmann, U.: Capturing variability in space and time with hyper feature models. In: Proc. Int'l Workshop on Variability Modelling of Software-Intensive Systems (VaMoS), pp. 1–8 (2014)

[171] Santos, I.S., Rocha, L.S., A. Santos Neto, P., Andrade, R.M.C.: Model verification of dynamic software product lines. In: Proc. of the 30th Brazilian Symposium on Software Engineering, SBES 2016, Maringá, Brazil, Sep. 19 - 23, 2016, pp. 113–122. ACM (2016). https://doi.org/10.1145/2973839.2973852

[172] Pnueli, A.: The temporal logic of programs. In: 18th Annual Symposium on Foundations of Computer Science (Sfcs 1977), pp. 46–57. https://doi.org/10.1109/SFCS.1977.32 . https://ieeexplore.ieee.org/abstract/document/4567924/authors#authors Accessed 2024-03-06

[173] Thüm, T., Batory, D., Kastner, C.: Reasoning about edits to feature models. In: Proc. Int'l Conf. on Software Engineering (ICSE), pp. 254–264 (2009). IEEE

[174] Dashjr, L.: BIP20: URI Scheme. https://bips.dev/20/

[175] Amick, S.: Understanding Taproot In A Simple Way. https://bitcoinmagazine.com/technical/understanding-taproot-in-a-simple-way Accessed 2025-07-18

[176] Chow, A.: BIP373: MuSig2 PSBT Fields. https://bips.dev/373/

[177] Greiner, S., Schultheiß, A., Bittner, P.M., Thüm, T., Kehrer, T.: Give an inch and take a mile? effects of adding reliable knowledge to heuristic feature tracing. In: Proc. Int'l Systems and Software Product Line Conf. (SPLC), pp. 84–95 (2024)

[178] Linsbauer, L., Schwägerl, F., Berger, T., Grünbacher, P.: Concepts of variation control systems. J. Systems and Software (JSS) **171**, 110796 (2021)

[179] Dit, B., Revelle, M., Gethers, M., Poshyvanyk, D.: Feature location in source code: a taxonomy and survey. J. Software: Evolution and Process **25**(1), 53–95 (2013)

[180] Xia, B., Bi, T., Xing, Z., Lu, Q., Zhu, L.: An Empirical Study on Software Bill of Materials: Where We Stand and the Road Ahead. In: Proc. Int'l Conf. on Software Engineering (ICSE), pp. 2630–2642 (2023)

[181] Agh, H., Azamnouri, A., Wagner, S.: Software product line testing: a systematic literature review. Empirical Software Engineering **29**(6), 146 (2024)

[182] Varshosaz, M., Al-Hajjaji, M., Thüm, T., Runge, T., Mousavi, M.R., Schaefer, I.: A classification of product sampling for software product lines. In: Proc. Int'l Systems and Software Product Line Conf. (SPLC), pp. 1–13 (2018)

[183] Noran, O.: Achieving a sustainable interoperability of standards. Annu. Rev. Control. **36**(2), 327–337 (2012)

[184] Ray, S.R., Jones, A.T.: Manufacturing interoperability **17**(6), 681–688 (2006)

[185] Sartipi, K., Yarmand, M.H.: Standard-based data and service interoperability in eHealth systems. In: Proc. Int'l Conf. on Software Maintenance (ICSM), pp. 187–196. IEEE (2008)

[186] Butler, S., Gamalielsson, J., Lundell, B., Brax, C., Mattsson, A., Gustavsson, T., Feist, J., Lönroth, E.: Maintaining interoperability in open source software: A case study of the apache pdfbox project. J. Systems and Software (JSS) **159** (2020)

[187] Lewis, G.A.: Role of standards in cloud-computing interoperability. In: HICSS, pp. 1652–1661. IEEE (2013)

[188] Laar, P., Hendriks, T.: A retrospective analysis of teletext: An interoperability standard evolving already over 30 years. Adv. Eng. Informatics **26**(3), 516–528 (2012)

[189] Siegmund, J., Siegmund, N., Apel, S.: Views on internal and external validity in empirical software engineering. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, vol. 1, pp. 9–19 (2015). IEEE