Architecture Design of Blockchain-Based Applications

Maximilian Wöhrer, Uwe Zdun
University of Vienna, Faculty of Computer Science
Vienna, Austria
{maximilian.woehrer,uwe.zdun}@univie.ac.at

Stefanie Rinderle-Ma
Technical University of Munich, Department of Informatics
Munich, Germany
stefanie.rinderle-ma@tum.de

***Abstract*—Integrating blockchain into software solutions is not straightforward as it requires sophisticated architectural design to connect and orchestrate centralized elements, such as backend logic, with decentralized elements, such as blockchain ledgers and smart contracts. We systematically explore this design space and possible architectural solution approaches. More specifically, we provide architectural blue prints for applications with different degrees of decentralization, describe conceptional components as well as possible relations between them.**

Our research shows that an event-driven architecture incorporating a messaging framework, tethered to dedicated components for handling blockchain state-changing and state-collecting operations, is a prevalent approach for choreographing blockchain-dependent business logic in blockchain-based applications.

***Index Terms*—blockchain, software architecture, decentralized application, DApp, smart contract, design pattern**

I. Introduction

A blockchain is a distributed and decentralized ledger that contains connected blocks of transactions [1] and can be thought of as a cryptographically secure transactional singleton machine with shared-state [2]. The technology allows the implementation of new software architectures for decentralized record-keeping and computation without relying on a (traditional) central point of trust. This aspect is beneficial in many cross-organizational processes [3] and efforts are being made to integrate the technology into enterprise applications. However, the acceptance and adoption of the technology in practical applications is still at an early stage. Accordingly, there is a lack of established design principles and design approaches that drive the integration of the technology into applications [4]. To address this gap, we follow up on previous research [5] that presents a general overview of architectural design decisions. In this context, we now turn to more detailed decision options as well as typical component structures, which we derive from existing architectural design solutions using grounded theory (GT) techniques to extract and identify common practices. If one considers the blockchain as a part of a larger system, it can be assumed that certain practices and architectures occur more frequently and thus prove to be more advantageous than others. Furthermore, we investigate existing, well-proven software design concepts and assess their applicability to blockchain-based applications.

In order to concretize the research objectives, we ask the following research questions: *RQ1)* Which existing architectural software design principles and concepts are suitable for blockchain-based applications and how can they be applied? *RQ2)* Which conceptual components in terms of architectural design exist and how are they related? For illustrative purposes, this paper primarily refers to permissionless blockchains, in particular Ethereum, today's most popular ecosystem. Please note that some concepts presented have a different valence in the context of permissioned blockchains (e.g., data confidentiality), but still remain applicable.

The paper is structured as follows: First, we discuss related work in Section II and our research methodology in Section III. Then, we elaborate the architectural design of blockchain-based solutions as main contribution in Section IV. Finally, we discuss findings in Section V and conclusions in Section VI.

II. Related Work

Blockchain-Oriented Software Engineering (BOSE) is dedicated to defining and applying software engineering principles and practices for blockchain-based system design, development, and deployment. Porru *et al.* [4] present one of the first works to identify issues, challenges, and peculiarities in this field. They advocate the need for new research directions and novel specialized blockchain software engineering practices. Wessling *et al.* [6] argue that a blockchain-oriented view is required for the architectural design process and propose the idea of blockchain tactics as a means to support the process of integrating decentralized elements in software architecture. In this work, however, the authors focus on the effects of design patterns at the implementation level and do not provide architectural guidance. Marchesi, Marchesi, and Tonelli [7] [8] propose a holistic agile software development process for gathering and analyzing requirements and designing, developing, testing, and deploying blockchain applications. Nonetheless, the approach is not specific enough to derive decisions on the architectural level. Udokwu, Anyanka, and Norta [9] explore and evaluate several high-level design approaches for developing blockchain-based applications, including the former two works. They propose another model-driven design framework with an automatic architecture model derivation, which lacks a detailed description. Bodkhe *et al.* [10] present various blockchain-based solutions and their applicability in various

Industry 4.0-based applications. In the aforesaid work, a blockchain-based reference architecture is described, but rather on a high level. Viswanathan, Dasgupta, and Govindaswamy [11] present a Blockchain Solution Reference Architecture (BSRA) that guides architects in creating end-to-end solutions based on Hyperledger Fabric. Architectural components are mentioned, but only described within a layered structure, so that the interaction of the components is not apparent.

So far, works that provide systematic architectural guidance in the field of BOSE are scarce. It is the goal of our work to remedy this shortcoming.

III. Research Study Design

In the search for (best) practices (hereinafter conceptually equivalent to software design patterns and other similar best practices), we apply a research methodology that is guided by the pattern derivation approach of Riehle, Harutyunyan, and Barcomb [12]. The approach describes the application of established scientific research methods for the purpose of pattern discovery and validation. In accordance with this approach patterns are discovered ("mined") and codified ("written") using Grounded Theory (GT) [13], [14] techniques. Driven by our research questions and known practices from our own experience, we defined initial search terms that were used to query major search engines (e.g., Google, Bing) in order to compile a number of well-fitting, technically detailed sources from the so-called "gray" literature [15] (e.g., practitioner reports, practitioner blogs, system documentation etc.). The resulting sources pool [16] was then examined in a later analysis with GT techniques. This included a thorough study and the annotation of the materials with labels ("codes" established with so-called "open coding") along with optional memos explaining important aspects of codes. Further, conceptual relations between codes (so-called "axial coding") were established to identify candidate categories for patterns. While this may indicate a simple linear execution of the work, pattern discovery and validation proceeded incrementally in several iterative stages, in which new sources (inspired from previous iterations) were exploited to constantly compare, revise, and contrast patterns until a theoretical saturation was reached. Theoretical saturation [13], [14] refers to a state in which adding new sources no longer yields new findings, and is commonly used as a stop criterion in GT-based studies.

IV. Architecture Design of Blockchain-Based Applications

Today, the design and development of applications based on blockchain technologies is a difficult undertaking and the degree to which the technology is used is also significantly influenced by characteristics such as performance, usability, and user experience. A well thought-out architectural design helps to balance these criteria. To this end, this section discusses design guidance for blockchain integration that we found and coded in our study. Architectural design options including practices/patterns, along with typical conceptual components and their relationships, are discussed and finally summarized in Table I Last, we discuss relevant topics in context such as microservices and Blockchain as a Service (BaaS).

A. Event-driven Architecture

As a software component, the blockchain has an asynchronous and event-driven character. This is due to the latency in the execution and confirmation of transactions and the fact that significant changes or operations that occur on the blockchain are usually propagated as events. Examples are events resulting from the execution of a smart contract or the creation of a new block. Given these characteristics, blockchains are not suited for real-time based systems and likewise scenarios where end-users expect an immediate impact of an operation. As with other systems that do not rely on synchronous communication (i.e., no strict arrival times of messages or signals), message coordination can be achieved by using event-driven architecture (EDA). Event-driven architecture is an architectural style in which there is no centralized controller to manage a workflow. Instead, different components interact with each other much more dynamically when certain events that affect their respective domains occur.

1) Event Sourcing: Event sourcing is a persistence concept used in event-driven architectures. It refers to storing application state as a sequence of immutable events. With it, a complete replay of the events that have happened since the beginning of the event recording can be achieved. This contrasts the traditional create, read, update, and delete (CRUD) approach, where only the current state of an object is stored and iteratively mutated. Event sourcing has several benefits. It allows for the creation of any number of user-defined data stores as materialized views of persisted events and knowledge about the state of domain objects at any given time by examining retroactive events. Blockchain and event sourcing share characteristics which suggest a natural affinity. Both share the concept of an "immutable append only log" which is considered as the single source of truth containing all events that have happened. Therefore, it seems natural to map, combine, and extend blockchains by event sourcing within application scenarios. In this sense, during our research we encountered use cases where blockchain was used as a trustless event store (e.g., [17]), and vice versa, where blockchain transactions were stored in a traditional event store (e.g., [18]).

2) Command-Query Responsibility Segregation: The Command-Query Responsibility Segregation (CQRS) pattern [19] is quite often mentioned alongside with event sourcing, because when using event sourcing some form of CQRS emerges almost naturally. CQRS is a design solution that segregates operations that read data from operations that write data by using separate interfaces and persistence models. This approach promotes separation of concerns, as the distinction between write and read aspects can result in persistence models that are more aligned, maintainable, and flexible. Most of the complex business logic can go into the write model, while the read model can be kept relatively simple. Further, problems such as scaling read and write operations, using optimized data schemata, and securing authorized writes are easier to solve. The pattern can be utilized for blockchain integration (see [17], [20]), for example to account for the general discrepancy of write and read operations. The write model is represented by executing transactions and storing information on the blockchain ledger. The read model, on the other hand, is a locally synchronized replica or materialized view of the blockchain to achieve fast read performance and rich querying capabilities.

B. General Application Architectures

Blockchain can be used as a standalone platform to implement the entire application logic (based on smart contracts), or as an auxiliary tool in larger enterprise solutions. Figure 1 illustrates this aspect and contrasts a traditional 3-tier application design with the main decentralization styles, namely a fully decentralized and a hybrid blockchain-driven architecture, discussed below.

1) Fully Decentralized Applications: A decentralized application (DApp) is a software solution that builds on a distributed computing platform. A DApp typically consists of a Web frontend that issues direct calls to a decentralized backend infrastructure (i.e., the blockchain executing smart contracts incorporating the entire application logic). This structure resembles a two-tier client-server architecture, with no intermediate support required for operation. The front-end code, which can be written in any language, can be hosted on a central server or on a decentralized storage (e.g., IPFS). Through the latter a complete decentralization of the application is achieved.

2) Hybrid (Semi-Decentralized) Applications: Building fully decentralized applications based solely on distributed components can be a difficult undertaking due to current technical limitations and usability challenges. Therefore, the current approach to building such applications is more nuanced. Rather than relying solely on decentralized components, a hybrid architecture is often implemented in which centralized components are beneficially added. In this context, a traditional backend is still relevant and several reasons speak for its use, although it lowers the trust compared to purely decentralized applications.

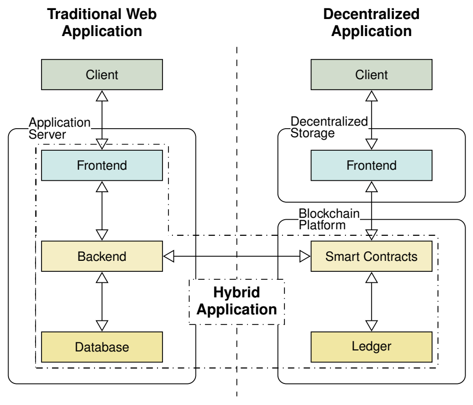

Fig. 1. A traditional, hybrid, and decentralized application architecture.

C. Relevance of the Backend

Blockchain is a closed ecosystem, thus smart contracts cannot directly interact with off-chain services to fetch information or trigger actions. On the contrary, they depend on the outside world to push information into the network or trigger actions by monitoring the network. This means a backend server is needed whenever a reliance on third-party services exists, such as ingesting external data, or performing mundane operations like sending emails. Another use for a backend is to act as cache or indexing engine for the blockchain, which also helps to provide a more responsive user interface and smoother perceived user experience. While the ultimate source of truth is the blockchain, clients can rely on the backend for search functions and validate the returned data on-chain. Next, large data storage is impractical on the blockchain due to the high costs associated with on-chain storage. Therefore, an application may need to rely on a backend to store large amounts of data, while only a hash is stored on the blockchain for validation. In the same way, complex calculations that would exceed the block gas limit of the blockchain can be moved to a backend. Another case where a backend can be useful is batching multiple transactions. The user can be relieved of repetitive transactions that take a lot of time by collecting user signed transactions in the backend and issuing them all at once. As long as these transactions are not time sensitive, batching them is a valid use case. A backend is also handy for automation, when a smart contract is designed to be called at a future time. Since smart contracts are passive entities, i.e. they do nothing until a participant explicitly interacts with them, there is no built-in ability to schedule events in smart contracts. Thus, a backend system can be used to reliably initiate periodic calls for smart contracts.

D. Conceptual Components

The anatomy of blockchain-based applications in terms of essential components is to some degree similar across different use cases. In the following, we list several components that every blockchain-like solution may wish to consider in its architectural design. Commonly found layout patterns of components are outlined using the example of a typical DApp incorporating a backend in Figure 2 and an enterprise-oriented application containing several loosely coupled services (e.g., microservices) in Figure 3. While there are many degrees of freedom in such architecture designs, such rough blueprint patterns can help in initial designs and for architecture classification. In the following, we discuss the typical conceptual components in those broader architecture patterns.

1) Wallet: Instead of a password-protected centralized account, blockchain users have a decentralized identity that is based on asymmetric encryption. The mechanism relies on pairs of keys: public keys, which may be distributed openly, and private keys, known only to the owner. A key pair is the identity on the blockchain. The public key (in a shorter representation) serves as account identification (address) and is derived from the private key that grants ownership of that

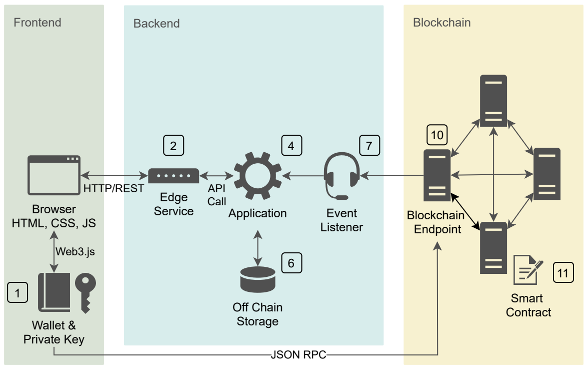

Fig. 2. Typical component structure pattern of a DApp.

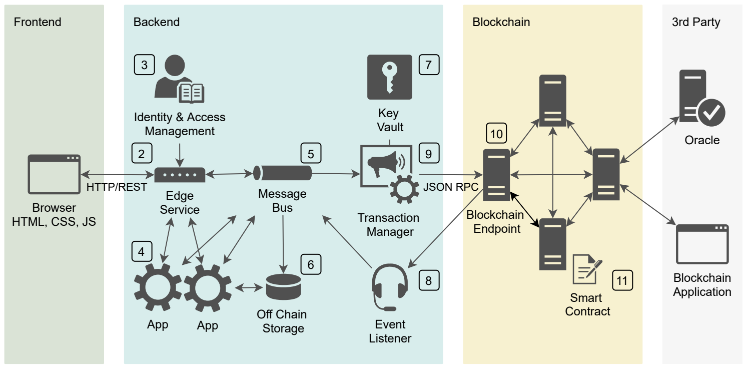

Fig. 3. Typical component structure pattern of an enterprise grade blockchain integration.

account. Therefore the private key is the most crucial information for identification and its safekeeping is essential. A wallet is either a device, physical medium, program or service that stores a user's private keys. In addition to this basic purpose, wallets often provide the functionality of encrypting, signing, and forwarding information (transactions) to the blockchain.

2) Edge Service: Edge services are components that are exposed to the public Internet and provide the capabilities required to deliver functionality and content to users over the Internet. They can refer to a multitude of components such as content delivery networks (CDNs), firewalls, load balancers, API gateways, reverse proxies, etc. They typically allow a shielded data flow from the Internet into the provider's infrastructure and into the enterprise. Edge services can also support backend applications by performing common tasks such as authentication, authorization, logging or monitoring.

3) Identity and Access Management: The identity and access management component stores user information to support user authentication and authorization as well as the provision of user data. Edge services can use this to control user-specific access to resources, services, and applications. For blockchain-based applications, it is necessary to align identity and access management holistically also with the blockchain inherent identity concept. It needs to be clarified to what extent users need sovereignty over their own blockchain identity and how this maps into an application-wide identity management perspective. For example, it is possible to leave the blockchain identity handling to the user or to the application as a custodian responsibility; in addition, an application-tailored blockchain identity concept without user binding is also conceivable. All of this must also be reconciled with transaction handling and secret key custody. In this context, it is possible to handle key management and transaction submission either entirely by the user or in the backend, or as a middle ground have meta-transactions where user-signed transaction requests are sent via a backend that pays for their execution.

4) Backend Application Logic: An application implements the logic required to achieve business objectives either by building a monolithic application, or a suite of small services organized by business capabilities that can be independently deployed. One of the most important considerations when integrating blockchain is what data and computations to put on-chain respectively off-chain. This decision is largely influenced by the business case and the intended benefit of using a blockchain (trust building, traceability, etc.) as well as current technological limitations. Generally, one should follow the basic design philosophy of using blockchains sparingly as they are slow and expensive. In terms of the communication flow between the application logic and the blockchain, this aspect is typically split into two separate components, one performing read (section IV-D8) and the other write (section IV-D9) operations. These components then serve as an interface to interact with the blockchain. When following a service-based design, it is also possible to further encapsulate both components through a dedicated "blockchain service" to have a central hub for blockchain interaction (e.g., Hyperledger Fabric Gateway).

5) Message Bus: In a service-oriented application, the services not only process requests from users, but interact with each other to handle these requests. Therefore, they must engage in an appropriate communication protocol. In such a situation, asynchronous communication by exchanging messages via a message framework has many advantages. The messaging framework takes the role of a message broker allowing to validate, store (buffer), transform, and route (one-to-one/many, content/topic-based) messages between services.

This approach offers the advantage of loose runtime coupling, because it decouples the message sender from the consumer. It also improves availability, given that the message framework buffers messages when a consumer is temporarily unavailable. However, this aspect also reveals a disadvantage, namely that the message framework must be highly available. Looking at our gray literature sources, the choice for a messaging framework basically comes down to message processing or stream processing. In message processing, messages are written to a queue and a broker takes care of delivering the published messages addressed to specific endpoints. Once the processing of a message is confirmed, the message is removed. This form of message delivery is suitable for environments with complex pre-definable and stable routing logic or where there is a need for guaranteed one-to-one delivery of messages. Common platforms for message processing are ActiveMQ or RabbitMQ. In stream processing, messages are written to a log that is persistent (limited to a retention period/size) and any endpoint may listen to these events and react accordingly. Messages are not removed once they are consumed, instead they can be replayed or consumed multiple times. This implies that the endpoints keep track of which messages to read (next). Popular stream processing platforms are Kafka or Pulsar. When using a streaming platform one can filter, aggregate, analyze or transform any blockchain events (e.g., mined transactions or blocks, emitted event logs) and also combine this information with non-blockchain events. Hence, one could build a streaming analytics process that performs state checks by monitoring contextually relevant events over time (e.g., [21]). To sum up, message processing is all about smart pipes, dumb endpoints; while data stream processing is the opposite: dumb pipes, smart endpoints. In principle, both messaging framework types can be used for blockchain integration, but this decision also depends on the surrounding application components and the integration effort regarding the software stack used. Overall, regardless of the taken approach, the logic implemented in endpoints should be idempotent, so that receiving the same event twice has no side effects.

6) Off-Chain Storage: Any data store outside the blockchain that holds data related to the blockchain can be considered as off-chain storage. Off-chain storage serves two main purposes. On the one hand, it should enable faster access to on-chain data through local replication, and on the other hand, it should decouple business data from the blockchain, be it for reasons of confidentiality or data size. The former is a read-only store and since it reflects the on-chain state, it should only be updated according to received blockchain events and not by business logic. Its purpose is to support caching and indexing to enable search, filter, sort, and pagination capabilities for on-chain data. There are several ways to realize this type of storage. For example, it is conceivable to use the messaging framework (e.g., Kafka) as event store and utilize its sink connectors (with Kafka Connect) to provide data to databases, key-value stores, and search indexes (e.g., [18]). For data provisioning from the blockchain, source connectors exist that allow to ingest blockchain data tapped via web3 into Kafka (e.g., [22]). Another approach is to create a separate data store and synchronization service that subscribes to various blockchain events on the message bus and pushes data to a storage, which later on is consumed by application services for queries. As a side note, there is also a decentralized solution for querying in which a frontend database is used. For this purpose a browser database (e.g., PouchDB, GunDB) syncs all relevant events, but this approach is not suitable for applications with a high data respectively event load. Now for the second purpose, namely a separate off-chain storage to detach business data from the blockchain. This type of storage can be used by business logic for a more controlled management of confidential data and may also serve as an exchange channel, if a shared storage is used. Its realization can take many forms depending on the type of data, such as a database (e.g., SQL, NoSQL) for metadata or a decentralized Content-Addressable-Storage (CAS) (e.g., IPFS, Swarm) for Binary Large Objects (BLOBs), whereby the integrity of the data is guaranteed by storing hashes on-chain. All things considered, it is advisable to treat both storage concepts separately, although it is technically possible to unify them. For completeness, if data is to be held only on-chain, techniques such as ordinary encryption, homomorphic encryption (HE), or zero-knowledge proofs (ZKPs) can be used to ensure data confidentiality.

7) Key Vault: A key vault is a component used to maintain control of encryption keys and other secrets. It is important for providing the private key in scenarios where transactions need to be signed in the backend. There are several complex strategies and different software solutions that allow storing private keys quite securely on the backend (e.g., HashiCorp Vault). Some solutions build on geographically distributed databases, while others built on specially designed hardware.

8) Event Listener: The event listener (e.g., Eventeum) is a service component in the backend infrastructure and listens for and reacts to events emanating from a blockchain system. It contributes to a clear separation of concerns by avoiding the need for services to subscribe directly to a blockchain endpoint for events. It handles (dynamically) registered event subscriptions, and broadcasts these events in a consumable manner (over a message bus) to downstream services running on the backend. Blockchain is a closed system, so for event retrieval, inevitably repeated polling against blockchain endpoints is required. Notably, there are two possibilities: Either make explicit endpoint protocol requests for certain events, e.g., to check whether a transaction has been mined based on a transaction hash, or follow a “crawler” based approach, where a bulk invocation retrieves all transactions at once given a (new) block number for examination. In both cases, in order to avoid undetected events, the event listener has to gracefully handle various errors that invariably occur in production systems: nodes out of sync, crashing nodes, congested nodes, network disconnects, stale data returned in requests, etc. This suggests that it is advantageous to use multiple (own) blockchain endpoints for redundancy. In this constellation an aggregator pattern can combine (and de-duplicate) multiple endpoint events to propagate the data in a reliable, at-least-once manner. In the same way, the event listener may consider the immutability of the event stream (depending on the consensus mechanism). For a consensus algorithm that allows multiple chain heads, there may be multiple competing event streams at any one time. It is either possible to wait for guaranteed event finality (sufficient succinct block confirmations), or propagate events as soon as they arrive and assume that downstream services handle ramifications of prematurely published events. The latter approach has been mentioned in a few gray literature sources and adopts an eventual consistency guided way of thinking for transactions, whereby the application assumes that any blockchain transaction being waited on, will eventually confirm and continues on as usual. This approach leaves the application in a state which is ahead of the blockchain, allowing for example an improved user experience. However, having two instances of state (i.e. blockchain and application) can be problematic if state management is not handled carefully including rollback scenarios; namely, when a transaction fails.

9) Transaction Manager: The transaction manager is a service within the blockchain application that receives messages and issues state-changing transactions (invoking smart contracts). It is an abstraction that controls how transactions are signed and broadcast to the blockchain network, via a connected blockchain endpoint. The component performs various tasks associated with the publication of transactions. First, it takes care of estimating adequate transaction costs, to ensure transactions are equipped with enough funds for a timely execution. Second, it takes care of nonce management. A nonce is an arbitrary (mostly sequential), unique number that is used to prevent replay attacks. Third, it deals with signing the entire transaction. This step usually integrates a key vault as a private key assembly solution. Some blockchain libraries (e.g., web3, ethers) embed the mentioned tasks behind the scenes, nevertheless it is important to know the background. In addition, the transaction manager has to handle various errors that may occur: nonce errors, network congestion, dropping peers, dropping transactions due to a sudden price increase, etc. In order to ensure reliable and stable transaction processing, the transaction manager can join forces with the event listener to verify that transactions are mined within a specified time. If this is not the case, a certain transaction can be republished with different parameters (e.g., corrected nonce, a higher tx fee) and monitoring starts again. It should be noted that there are API gateway service providers (e.g., EthVigil) that offer transaction lifecycle management and also enable blockchain monitoring via webhooks or websockets.

10) Blockchain Endpoint: A blockchain endpoint is a device or node running a piece of software that implements the blockchain protocol. A node verifies all transactions in each block, keeping the network secure and the data accurate. Typically, different node types and polyglot node implementations exist (e.g., Geth, Parity). A full node has the entire blockchain downloaded and available. Hence, it can verify transactions and execute smart contracts independently. A blockchain network is maintained and operated by full nodes. A light node only holds block headers and can validate transactions with the support of a full node. In addition, there are also service providers that operate node clusters (e.g., Infura, QuikNode) which allow users to interact with the blockchain without having to set up their own node. All in all, operating an own node requires no trust in the network since the data can be verified in the node itself. If the blockchain is to be used in a truly private, self-sufficient and trustless manner, the operation of an own node is required.

11) Smart Contract: A smart contract implements the business logic on the blockchain and can be seen as a self executing autonomous entity. As a design principle, contracts should be focused on a single responsibility or capability (preferring many simpler smart contracts to a few larger ones) and constructed to minimize the number/size of on-chain transactions/write operations (to reduce costs) and the dependencies required for testing. For common concerns (e.g., access control) audited and production-tested library contracts (e.g., OpenZeppelin) and standardized contract implementations (e.g., ERC-20) should be used. Further, it is advisable to conduct peer reviews that focus on reducing excessive complexity and to consult specialized smart contract auditors. Due to the inherent characteristics/limitations of blockchain-based program execution, smart contracts require a rather unconventional programming paradigm. To handle those challenges, various design patterns emerged. A detailed discussion is beyond the scope of this paper, thus we refer to [23] [24]. In terms of data processing, smart contracts have their own state, but mostly they operate in relation to a common data model within the domain of a system, thus modeling and handling state transfer is a main concern. Complex aggregate or inferred state computations are typically kept off-chain and pushed on-chain with trusted oracles as needed. Another topic worth mentioning is the structural design and layout of smart contracts. In this context, there are different design options: A single smart contract acts as an interface (facade) that orchestrates interaction with other downstream smart contracts, or multiple smart contracts act independently with equal priority, and their functionality is being combined within a client/backend. In certain scenarios it is also common to use a template (factory) contract on-chain to instantiate contracts with the same structure and flow, but for a different context.

TABLE I
DESIGN DECISIONS, OPTIONS, AND CONCEPTUAL COMPONENTS.

Design Decision	*Design Option*	*Conceptual Component*
Decentralization Level	Full	N/A
	Partial	N/A
Identity Provisioning	Blockchain	Wallet
	Custodial	ID & Access Mngmt
	In-House	
Transaction Handling	User-Tx	Wallet
	Meta-Tx	Transaction Manager
	Backend-Tx	
Key Management	User	Wallet
	Backend	Key Vault
Transaction State Sync	Strict	Event Listener / Backend App. Logic
	Eventual	
Blockchain Connection	Own Node	Blockchain Endpoint
	3rd Party	
Frontend Provisioning	Decentr. Storage	N/A
	Backend	Edge Service
Application Logic	On-Chain	Smart Contract
	Off-Chain	Backend App. Logic
Component Orchestration	Point-to-Point (Queue)	Message Bus
	Pub/Sub (Topic)	
Rich Querying	Frontend DB	N/A
	Backend DB	Off-Chain Storage
Confidential Storage	On-Chain (Encr., HE, ZKPs)	Smart Contract
	Off-Chain (CAS & Hash Ref.)	Off-Chain Storage

E. Blockchain Smart Contracts and Microservices

Microservices are an application architectural style in which a complex application is composed of many smaller, discrete, decoupled, and network-connected services that communicate with each other using standardized interfaces. Although smart contracts and microservices are fundamentally different in terms of the native environments they serve (decentralized vs centralized platforms) and the challenges they seek to address, they are both a response to the rise of distributed architecture. While smart contracts are more about enabling transactions in low-trust environments, microservices are about enabling modularity and scale. However, smart contracts and microservices also have commonalities from a service-oriented architecture perspective (see [25] [26] [27]). Both are designed for focused functionality, autonomy, composability, and communication via standardized and well-defined interfaces. Hence, smart contracts can to some extent be interpreted as services of a blockchain-based computing paradigm. In this light, it makes sense to combine both concepts and design blockchain-based applications with microservices architecture principles. Following this approach brings not only the benefits associated with microservices (e.g., loose coupling, scalability, polyglot development, etc.) but also facilitates blockchain integration. When blockchain is treated as a service component in a microservices paradigm, it becomes easier to deal with its asynchronous and event-based nature. Proven concepts in microservices architecture, such as EDA, event sourcing, and CQRS, provide useful tools in this context. With EDA blockchain transactions can emanate as events and the flow of information within a system can be organized asynchronously in coordination with these events. Event sourcing and CQRS complement this approach, as blockchain state changes can be stored as a continuum of immutable events within an event store, from which any view or structural model can be derived.

F. Blockchain as a Service (BaaS)

Looking at current trends in software development, one can speak of a new cloud-native application era where IT systems and applications are increasingly being outsourced to cloud service providers, for reasons of cost savings and improved management and maintenance. This trend has also caught up with the blockchain sector under the term Blockchain as a Service (BaaS). BaaS enables businesses to rely on a service provider to provision and manage aspects of a blockchain infrastructure in order to facilitate the development, testing, deployment, and ongoing management of blockchain applications. This approach allows development resources to be better focused on a specific goal by utilizing a wide range of readily available components while avoiding infrastructure and platform configuration overhead. A cloud environment can also be attractive in order to keep access and cooperation hurdles low for other participants (e.g., in a consortium). However, there are also disadvantages. Relying on a single service provider to run a decentralized blockchain network can be contradictory as it introduces a form of (re)centralization (trusting those who manage infrastructure). There is also the risk of a vendor lock-in, as it is difficult or very expensive to switch to a different service provider (due to lack of standardization especially for BaaS); the same also applies to the selection of a blockchain platform.

Many well-known IT companies such as Microsoft, Amazon, IBM, SAP, or Oracle now offer BaaS solutions (for a comparison see [28] [29] [30]) and enable the operation of blockchain nodes on their respective platforms, some even allow the use of third-party infrastructure. Various blockchain ecosystems and consensus mechanisms are offered, mostly geared towards permissioned blockchains (e.g., Hyperledger Fabric, ConsenSys Quorum) that focus on performance and speed with strict privacy and access controls, which are preferred for business collaboration. In addition, many platforms offer a variety of different applications for operations, node, and smart contract management. Some even offer ready-to-use templates to provide predefined blockchain network configurations or generic applications (e.g., supply chain, financial services). Pricing models vary (e.g., number of nodes/transactions, storage space required, CPU utilization), also among providers, and often tenants can choose among several different options. As for the degree of built-in blockchain integration features, these vary between service providers. Some providers allow a deep integration with other built-in platform services out-of-the-box (e.g. through adapters, connectors, triggers, purpose fitted SDKs, etc.) using different integration approaches (e.g., serverless functions, workflow orchestration), while others focus on infrastructure provision and basic interaction. With service providers in the first group, a Function-as-a-Service (FaaS) approach is feasible, allowing the use of serverless computation options based on event-driven models where a piece of code (aka "function") calling a smart contract is invoked by an event-based trigger such as a HTTP request (or any other event). This enables, for example, the implementation of a uniform interface for smart contract interaction in a REST-API manner.

V. Discussion

While it may seem straightforward to use blockchain for specific business scenarios, using the technology often comes at the expense of scalability, privacy, and usability. As a result, the architectural design of blockchain-based applications is currently influenced by the need to compensate for theses drawbacks. This is being achieved with a variety of approaches, all aimed at shifting responsibility to centralized components in areas where blockchain is currently lacking. This leads to hybrid architectures combining centralized and decentralized components, in which the engineering challenge is comprised of ensuring a smooth and timely communication between the two. In such environments, it can be advantageous to treat events as first-class citizens of an application and use a messaging framework to coordinate communication between different components in an event-driven manner. As far as communication with the blockchain is concerned, the interaction can be narrowed down to two aspects. First, listening to blockchain events and reading the blockchain network state. Second, publishing transactions to invoke state-changing operations. This is basically a two-way communication where the transaction manager is the write channel and the event listener is the read channel. By applying a service-oriented paradigm to these components and using asynchronous communication, blockchain integration boils down to choreograph blockchain-dependent business logic with these gateway services.

Setting up application and blockchain infrastructure components can be a time-consuming and laborious task. Instead of developing an in-house solution with self-managed software, it can be efficient to outsource the infrastructure challenges to API gateway services or to cloud service providers. While gateway services abstract and encapsulate blockchain interaction behind HTTP API calls, cloud service providers offer a comprehensive implementation platform with a set of architectural artifacts that can be leveraged to accelerate the development. Various compute, orchestration, storage, messaging, logging, and monitoring services can be combined. Service providers with built-in integration options also have the advantage of being able to abstract direct interactions with the blockchain, facilitating the flow of communication and eliminating concerns about the reliability of transaction handling and event notification. In addition, available integration options can offer a scaffolding, that allows to wrap blockchain interaction into easier consumable and integratable blockchain services (e.g., as FaaS).

Overall, BaaS solutions can ease blockchain and application infrastructure management and speed up development, but are not on par with self-managed solutions in terms of provided trust. Nevertheless, the degree of trust can be indirectly influenced by how much is managed by a service provider, respectively whether BaaS is consumed as SaaS, PaaS, or IaaS.

VI. Conclusion

Blockchain is a disruptive technology for enabling mutual trust in collaborative environments and has the potential to replace existing business models with new technological solutions. However, to leverage the technology, new types of architectures and designs are needed. To this end, we have described architectural design solutions for creating blockchain-based applications with different degrees of decentralization. In this context, we studied existing solutions from which we inferred typical architectural layouts and conceptual components. Based on our findings, we identified high-level architectural blueprints or patterns in which we described key components along with their purpose and interaction.

While identifying suitable business cases is key to the success of using blockchains, it is equally important to build solutions on a robust architecture. For the integration of blockchain with its asynchronous and event-driven character, it is natural to adopt architectural styles and programming paradigms that are focused on these very characteristics. In this regard, an event-driven architecture consisting of reactive components such as microservices or, in the cloud context, serverless functions, and dedicated blockchain read/write gateway services provides a good fit.

Although blockchain has not yet broken through as a mainstream application foundation today, the technology holds the potential to reshape business relationships once operational improvements and increased efficiencies come to fruition. Along the way, future research could investigate (architectural) standards for commonly occurring application scenarios to foster a quick adoption of blockchain technology.

References

[1] M. Samaniego and R. Deters, "Blockchain as a Service for IoT," in *Proc. - 2016 IEEE Int. Conf. Internet Things*, 2017, pp. 433–436.

[2] G. Wood, "Ethereum: a secure decentralised generalised transaction ledger," *Ethereum Proj. Yellow Pap.*, 2014.

[3] J. Mendling, I. Weber, W. Van Der Aalst, *et al.*, "Blockchains for business process management - Challenges and opportunities," *ACM Trans. Manag. Inf. Syst.*, 2018.

[4] S. Porru, A. Pinna, M. Marchesi, *et al.*, "Blockchain-oriented software engineering: Challenges and new directions," *2017 IEEE/ACM 39th Int. Conf. Softw. Eng. Companion*, no. February, pp. 169–171, 2017.

[5] M. Wöhrer and U. Zdun, "Architectural Design Decisions for Blockchain-Based Applications," in *3rd IEEE Int. Conf. Blockchain Cryptocurrency*, 2021.

[6] F. Wessling, C. Ehmke, O. Meyer, *et al.*, "Towards Blockchain Tactics: Building Hybrid Decentralized Software Architectures," in *2019 IEEE Int. Conf. Softw. Archit. Companion*, IEEE, 2019, pp. 234–237.

[7] M. Marchesi, L. Marchesi, and R. Tonelli, "An Agile Software Engineering Method to Design Blockchain Applications," pp. 1–8, 2018.

[8] L. Marchesi, M. Marchesi, and R. Tonelli, "ABCDE -Agile Block Chain dApp engineering," no. December, 2019.

[9] C. Udokwu, H. Anyanka, and A. Norta, "Evaluation of Approaches for Designing and Developing Decentralized Applications," no. June, pp. 1–18, 2020.

[10] U. Bodkhe, S. Tanwar, K. Parekh, *et al.*, "Blockchain for Industry 4.0: A comprehensive review," *IEEE Access*, vol. 8, pp. 79 764–79 800, 2020.

[11] R. Viswanathan, D. Dasgupta, and S. Govindaswamy, "Blockchain Solution Reference Architecture (BSRA)," pp. 1–12, 2019.

[12] D. Riehle, N. Harutyunyan, and A. Barcomb, "Pattern Discovery and Validation Using Scientific Research Methods," no. February, 2020.

[13] B. G. Glaser and A. L. Strauss, *Discovery of grounded theory: Strategies for qualitative research*. 2017.

[14] J. M. Corbin and A. Strauss, "Grounded theory research: Procedures, canons, and evaluative criteria," *Qual. Sociol.*, 1990.

[15] V. Garousi, M. Felderer, M. V. Mäntylä, *et al.*, *Benefitting from the grey literature in software engineering research*, 2019.

[16] Arch. Design of BC-Based Applications - Knowledge Sources, [Online]. Available: https://github.com/maxwoe/bc_architecture_design.

[17] Transmute Framework, [Online]. Available: https://github.com/transmute-industries/transmute.

[18] Ocean Bounty: Smart Contract Event Monitoring Tool, [Online]. Available: https://explorer.bounties.network/bounty/2146.

[19] C. Richardson. (2017). Microservices Patterns, [Online]. Available: https://microservices.io/patterns/index.html.

[20] Hyperledger Fabric CQRS-ES, [Online]. Available: https://data.hkoscon.org/event/make-hyperledger-fabric-reactive-and-cqrs-es.

[21] Blockchain Streaming Analytics, [Online]. Available: https://www.youtube.com/watch?v=rY1fEaCvwXk.

[22] Kafka-web3-connector, [Online]. Available: https://github.com/satran004/kafka-web3-connector.

[23] X. Xu, C. Pautasso, L. Zhu, *et al.*, "A pattern collection for blockchain-based applications," in *ACM Int. Conf. Proceeding Ser.*, 2018.

[24] M. Wöhrer and U. Zdun, "Design Patterns for Smart Contracts in the Ethereum Ecosystem," in *2018 IEEE Int. Conf. Internet Things*, 2018, pp. 1513–1520.

[25] I. Weber, "Blockchain and Services – Exploring the Links: Keynote Paper," *Lect. Notes Bus. Inf. Process.*, vol. 367, no. October 2019, pp. 13–21, 2019.

[26] F. Daniel and L. Guida, "A Service-Oriented Perspective on Blockchain Smart Contracts," *IEEE Internet Comput.*, vol. 23, pp. 46–53, 2019.

[27] G. Falazi, A. Lamparelli, U. Breitenbuecher, *et al.*, *Unified Integration of Smart Contracts through Service Orientation*, 2020.

[28] M. M. H. Onik and M. H. Miraz, "Performance Analytical Comparison of Blockchain-as-a-Service (BaaS) Platforms," *Lect. Notes Inst. Comput. Sci. Soc. Telecommun. Eng. LNICST*, vol. 285, pp. 3–18, 2019.

[29] A. Kernahan, U. Bernskov, and R. Beck, "Blockchain out of the Box – Where is the Blockchain in Blockchain-as-a-Service?" *Proc. 54th Hawaii Int. Conf. Syst. Sci.*, vol. 0, pp. 4281–4290, 2021.

[30] V. Yussupov, G. Falazi, U. Breitenbücher, *et al.*, "On the serverless nature of blockchains and smart contracts," *arXiv*, 2020.