AstraKode-Katena: Democratizing Advanced Technologies through Low-Code Tools

Davide Gigante, Federica Topazio
*Dipartimento di Elettronica, Informazione e Bioingegneria, Politecnico di Milano, Milan, Italy

***Abstract*—Blockchain technology, despite its revolutionary potential, faces significant barriers to adoption due to its inherent complexity and the specialized knowledge required for its implementation. This paper presents the integration of AstraKode, a low-code development platform, with the Katena framework, a model-driven engineering approach, to simplify the creation, deployment, and management of blockchain applications. By combining AstraKode's intuitive, visual interface for smart contract development with Katena's declarative, high-level abstraction for blockchain infrastructure, this integration lowers the technical entry barriers, enabling a broader audience to leverage blockchain technology. A case study involving the development of a Non-Fungible Token marketplace demonstrates the effectiveness of this approach. The marketplace is built using AstraKode's visual tools and deployed via Katena, showcasing how this combined framework can streamline blockchain application development, enhance scalability, and ensure robust lifecycle management. The results highlight the potential of low-code platforms to democratize access to advanced blockchain technologies, making them more accessible to users with limited technical expertise**

***Index Terms*—**

I. Introduction

Blockchain has emerged as a disruptive technology based on a decentralized and traceable ledger, allowing users to securely record transactions in a trust-less environment with hard security guarantees[Jav+22]. Initially conceptualized as the underlying technology for Bitcoin, but now it has evolved significantly and it serves as the foundation for various applications beyond cryptocurrencies. A blockchain consists of a chain of blocks, each containing a list of transactions[Zhe+18]. These transactions are verified and added to the blockchain by a network of nodes through a consensus mechanism, ensuring the integrity and security of the data[Li+18].

The decentralized nature of blockchain eliminates the need for intermediaries, hence reducing transaction costs and facilitating efficiency in peer-to-peer transactions[CB20]. Each participant in the blockchain network maintains a copy of the ledger, and all changes are visible to everyone, ensuring transparency. The security of blockchain is reinforced by cryptographic algorithms that make it nearly impossible to alter the data

once it is recorded[NR19]. This feature is particularly valuable for applications requiring a high level of trust and integrity, such as financial services, supply chain management, and healthcare[Has+20]. However, sometimes the impossibility to change something in a node can be a limit.

Many blockchains have become more like computers by adding the ability to create smart contracts. These smart contracts are flexible programs that run on the blockchain[Sta+17]. Ethereum is one of the most important platforms for blockchain applications, and its runtime environment, the Ethereum Virtual Machine (EVM), has been adopted by several other blockchains[Hil+17]. However, the complexity of blockchain infrastructure and the need for specialized knowledge in cryptography, distributed systems, and programming present substantial barriers to entry[PPP20]. Developing blockchain applications typically involves writing smart contracts—self-executing[Moh18] contracts with the terms of the agreement directly written into code—which requires proficiency in languages such as Solidity for Ethereum-based platforms.

Despite its advantages, blockchain technology also has a lot of significant challenges, particularly in terms of accessibility and usability. The technical background needed to develop a blockchain technology, creates a limit in the adoption of these technologies to those with advanced technical skills, thus excluding a broader audience from leveraging its potential[Bra+20]. Additionally, the rapid pace with which blockchain is experiencing a development means that developers must continuously update their knowledge and skills to keep up with new advancements and security practices[San+21]. This dynamic nature of blockchain technology can be daunting for many businesses and individuals looking to integrate it into their operations.

Adopting a declarative approach could enhance the efficiency of Blockchain operations and offer a unified method for managing hybrid applications—those where the application logic is executed both on and off the Blockchain [Bar+22]. Here, the Katena framework [Bar+22], a declarative framework for the deployment and management of Ethereum and EVM-compatible applications, comes into play. KATENA features a model-driven engineering (MDE) methodology. Katena provides a high-level abstraction over the underlying blockchain infrastructure, allowing developers to focus on application logic rather than the intricacies of blockchain implementation.

Katena offers a metamodel that defines reusable components for creating application models. These components include infrastructural nodes, smart contracts, and other off-chain elements. By embedding lifecycle operations within each component, Katena simplifies the deployment and management of blockchain applications. Developers can model decentralized applications (dApps) through the instantiation and composition of these components, streamlining the development process and reducing the technical expertise required.

However, the creation of blockchain-based software applications still requires considerable technical knowledge, particularly in software design and programming. This is regarded as a major barrier to adopting this technology in business and making it accessible to a wider audience. As a solution,

low-code and no-code approaches have been proposed that require little or no programming knowledge for creating full-fledged software applications, such as the Astrakode tool [CHF23].

The complexity of writing smart contracts and managing blockchain infrastructure can be further mitigated by integrating Katena with a low-code development environment. Low-code tools enable users to create applications through graphical interfaces with minimal coding, making technology accessible to a wider audience. One such tool is the Smart Contract IDE developed by AstraKode, which provides a visual environment for designing and deploying smart contracts.

By combining Katena's declarative framework with AstraKode's low-code tool, developers can leverage the strengths of both platforms. The visual interface of the Smart Contract IDE allows users to design smart contracts without needing in-depth programming knowledge, while Katena handles the underlying blockchain infrastructure and lifecycle management. This integration bridges the gap between high-level application design and low-level blockchain implementation, making blockchain technology more accessible and user-friendly.

An illustrative use case of this integrated approach is the implementation of a Non-Fungible Token (NFT) Marketplace[Wan+21]. The aim was to develop the aforementioned marketplace on the platform, implementing it using the visual tool. Moreover, the main focus of the study was to understand the feasibility of integrating Astrakode with Katena, to close the gap between contracts in Solidity/JSON and the visual creation of them on the tool.

II. Background

The following sections introduce the background concepts required to fully understand what is proposed in the following paper: Blockchain, NFT, No Code Tools and Katena. The reader familiar with these concepts can easily skip or skim through them.

A. Blockchain

A Blockchain is a decentralized peer-to-peer network consisting of nodes that maintain a ledger of user transactions. These transactions are organized in an append-only sequence of blocks, forming a chain, and collectively result in a shared state derived from all executed transactions. A transaction represents a set of user-initiated actions that alter this shared state. Blockchains typically feature a native currency (e.g., ether in Ethereum), allowing transactions to be purely monetary (e.g., Alice sends 10 ether to Bob) or computational, involving smart contracts that require payment for execution.

Blockchains address the Byzantine Generals Problem by enabling the validation of a shared state in a trustless, decentralized environment. Each node holds a local copy of the Blockchain (ledger, state, and smart contracts) to validate new transactions. The decentralized consensus mechanism, relying on cryptography and incentives, ensures nodes agree on valid transactions, which are then bundled into new blocks. Honest participants are rewarded in the native currency for their efforts.

To execute transactions, users need a wallet—a program that interacts with the Blockchain. A wallet includes a public key, serving as a public address, and a private key, enabling transaction creation. Many Blockchains support smart contracts, which

are computer programs stored and executed within the Blockchain. Some frameworks, like Bitcoin Script, offer simplified, non-Turing-complete languages, while others, like Ethereum and EVM-compatible networks, support general computation using languages like Solidity and Vyper, or more familiar languages such as Rust and JavaScript.

Smart contracts have an object-like structure, including a constructor, state variables, and functions with visibility modifiers. They can utilize libraries—special contracts with no internal state but reusable functions callable by other contracts. In Ethereum and EVM-compatible blockchains, smart contracts and libraries are compiled into bytecode executable by the Ethereum Virtual Machine (EVM). An Application Binary Interface (ABI) is generated to facilitate interactions between on-chain and off-chain components.

Deployment involves sending a transaction containing the bytecode to the network. Upon acceptance, the bytecode and contract state are stored at a specific Blockchain address. Each library is deployed separately with its address. Smart contracts can be deleted, freeing storage and providing a reward as an incentive. The execution cost of a smart contract, paid in the native currency, prevents network flooding and mitigates Denial of Service attacks. Only users can initiate smart contract execution via transactions, allowing contracts to call other contracts within ongoing transactions.

B. NFT

Non-fungible tokens (NFTs) are unique digital assets such as pieces of art, digital content, or videos that have been tokenized using a blockchain. The tokens are created through an encryption function that generates unique identification codes from metadata. These tokens are stored on a blockchain, while the actual assets are stored elsewhere. The unique connection between the token and the asset distinguishes NFTs from other digital assets.

NFTs can be bought, sold, and exchanged for money, cryptocurrencies, or other NFTs [Hay23]. Unlike fungible assets, which are interchangeable, NFTs have distinct values and identities. Cryptocurrencies like Ether are fungible tokens because each token has the same value and function. In contrast, each NFT is indivisible and unique, meaning it cannot be split or merged with other tokens [Che18].

The Ethereum Blockchain's first token standard, ERC-20 (Ethereum Request for Comment), supports only fungible tokens [FV15]. ERC standards are predefined rules developed using smart contracts for implementing tokens on the Ethereum Blockchain. These tokens are linked to digital objects through metadata, which facilitates off-chain rendering or storage [MSS20].

C. No-Code Tools

The creation of blockchain-based software applications requires today considerable technical knowledge, particularly in software design and programming. This is regarded as a major barrier in adopting this technology in business and making it accessible to a wider audience. As a solution, low-code and no-code approaches have been proposed that require only little or no programming knowledge for creating full-fledged software applications.[CHF23]

The AstraKode Blockchain platform is an appropriate low-code platform for enterprise

blockchain solutions. It provides Network Composer for customization of blockchain networks, a visual environment, smart Contract IDE and cloud deployment provisions. It provides native support for the most common permissioned blockchains, custom networks and smart contracts may be designed and developed with speed and ease, the capacity to design, build, and deploy a production grade solution and its low-code strategy facilitates project self-documentation and validation.[Nav+23]

D. Katena

KATENA is a declarative framework for the deployment and management of Ethereum and EVM-compatible applications. KATENA features a model-driven engineering (MDE) methodology [BCW12] at its core. A metamodel defines a set of reusable components for the creation of application models, ranging from infrastructural nodes to smart contracts, and other components that run outside the Blockchain (i.e., off-chain). Lifecycle operations (i.e., how to deploy and update a single component) are embedded in each component, to further ease the modeling task. KATENA also provides a set of processes to enact the deployment and management of defined models in a sound and automated way. KATENA is based on a metamodel that defines a set of resuable on- and off-chain components to model dApps: users create application models by instantiating and composing them. A decentralized application (dApp) is a software application that runs on a decentralized network, typically a blockchain, rather than on a single centralized server.[Bar+22] Katena enables developers to efficiently model dApps through instantiation and composition, emphasizing lifecycle management and dependency handling. Additionally, Katena creates a visual and functional map of the dependencies among components, dictating the order of management and deployment through a dependency graph (Dependency graph).

KATENA categorizes dependencies into five types, each influencing dApp deployment and operation differently. The first type involves dependencies between libraries and either other libraries or smart contracts. These dependencies require embedding the library's address into the contract's bytecode, initially using placeholders that are updated post-deployment.

The second type covers dependencies between smart contracts themselves. In standard contract-to-contract dependencies, a contract must store the address of another contract at its creation, enforcing a sequential deployment order. However, a more flexible approach, known as lazy contract-to-contract dependency, allows contracts to be associated via dedicated functions after deployment. This enables parallel deployment, with the necessary connections established through subsequent function calls.

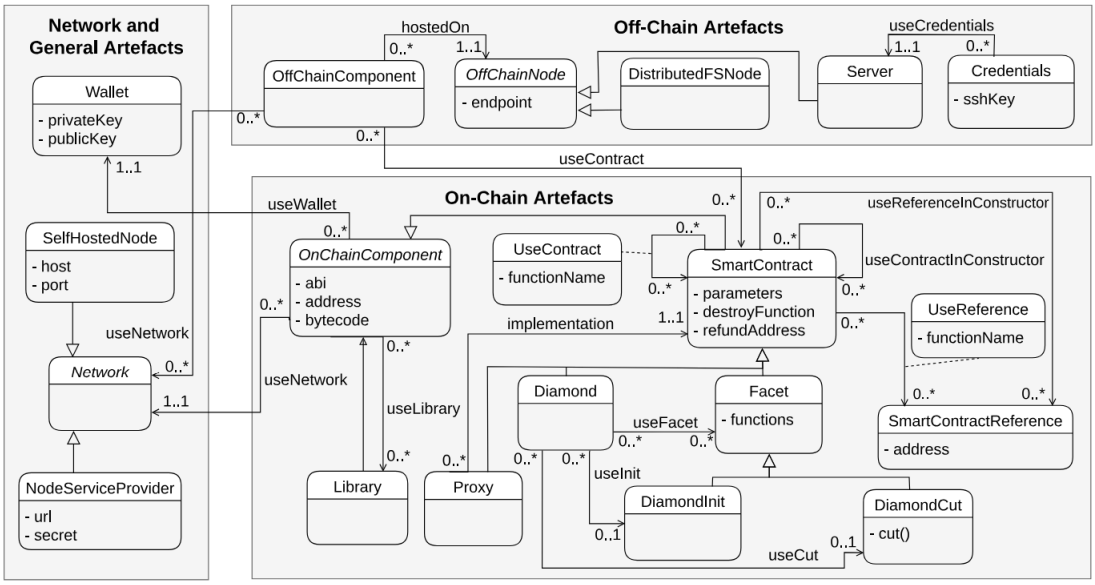

Fig. 1. KATENA Metamodel[Bar+22]

Finally, there are off-chain to on-chain dependencies, which deal with the interactions between off-chain components and on-chain smart contracts. These dependencies are crucial for setting up communication between external systems and the blockchain. They require the smart contract's address and a blockchain endpoint to be known, meaning the deployment of off-chain components must wait until the on-chain contracts are deployed.[Bar+22]

The KATENA metamodel, depicted as a UML class diagram, categorizes its elements into network and general artifacts, on-chain artifacts, and off-chain artifacts. All attributes are private, with omitted getters and setters for brevity. Internal methods manage component lifecycles and enforce constraints, such as preventing circular dependencies between smart contracts. The prototype implementation in Section 4 checks these constraints and aborts deployment if violated.

Network and General Artifacts represent how applications and users interact with a blockchain. The core entity, Network, provides entry points for blockchain operations, with subclasses SelfHostedNode and NodeServiceProvider catering to different blockchain access setups. SelfHostedNode includes attributes like host and port for connecting to blockchain nodes, including local testing environments like Ganache. NodeServiceProvider interacts with the blockchain via a URL and an access key. User wallets, modeled separately, store the public and private keys necessary for transaction management and smart contract deployment.

On-chain Artifacts are assets deployed on the blockchain, modeled by the abstract class OnChainComponent. This class includes attributes like the ABI (known before deployment) and bytecode (which may update dur-

ing deployment due to dependencies). SmartContract entities define constructor parameters, which may include other smart contract addresses, modeled as useReferenceInConstructor or useContractInConstructor associations. SmartContract also includes attributes for destruction functions and refund addresses, and associations for managing lazy dependencies post-deployment. Libraries, modeled by the Library class, do not have state or constructors and cannot be removed, but can be connected to other components.

Upgradeable Smart Contracts are represented by Proxy and Diamond entities. Proxy abstracts the ERC1967 standard, referencing underlying contracts with application logic. Diamond contracts, part of the diamond pattern, connect to Facet instances through useFacet associations, supporting lazy dependencies. DiamondCut and DiamondInit classes, inheriting from Facet, are linked to Diamond for specific functions.

Off-chain Artifacts are modeled by the abstract OffChainComponent class, which connects to blockchain networks and interacts with multiple smart contracts. These components require smart contracts' ABIs for interaction, typically included in the source code. OffChainComponent can be hosted on various infrastructures, linked to OffChainNode through the hostedOn association. OffChainNode specializes into DecentralizedStorage for f-dApps and Server for h-dApps, with Server nodes further linked to credentials for SSH connections.

This structured approach ensures comprehensive modeling of both on-chain and off-chain elements, supporting various deployment and operational scenarios within the KATENA framework.

The proposed metamodel in KATENA allows users to construct their dApp application model through the composition of its elements, without needing to manage the processing mechanisms of the entire model. KATENA introduces two processes for handling the deployment and management of Blockchain applications, enabling users to operate at a higher abstraction level. Users only need to instantiate the necessary components and their dependencies.

The first process involves deploying a Blockchain application. It begins by setting up connections with both on- and off-chain infrastructures through subclasses of Network and OffChainNode. A dependency graph is then created by traversing relevant associations in the metamodel, such as useContractInConstructor and useLibrary. This graph guides the deployment sequence, starting with components that have no dependencies and progressing to those that do.

Initially, KATENA deploys independent libraries and retrieves their addresses to resolve L-L dependencies. It then deploys smart contracts, starting with those without dependencies, followed by those with C-L dependencies, and finally those with contract-related dependencies, ensuring all on-chain components are deployed. Afterward, Lazy-C-C dependencies are resolved, completing the smart contract setup. Once on-chain components are deployed, off-chain components are instantiated using the necessary Blockchain endpoints and contract addresses.

The second process focuses on managing and upgrading deployed components, which may involve redeploying existing ones. When upgrading a component, KATENA identifies the node to be replaced and initiates the deployment of its upgraded version. For li-

braries, it resolves L-L and L-C dependencies by redeploying dependent libraries and smart contracts with updated addresses. For smart contracts, Lazy-C-C and C-C dependencies are managed, either by updating the dependency dynamically or by redeploying the smart contract with the new address. Finally, KATENA manages O-O dependencies by updating the addresses of the on-chain components where necessary.

This structured approach streamlines the deployment and management of Blockchain applications, ensuring that dependencies are systematically resolved and components are efficiently upgraded.[Bar+22]

III. AstraKode Blockchain

The Astrakode tool employs a model-driven approach to abstract both blockchain networks and smart contracts. This is accomplished by developing ontologies through diagrams, which are then formally defined using XML files that adhere to established baselines such as metamodels and grammars. These ontologies are subsequently translated into a format that can be rendered by the front-end engine, utilizing a meta-ontology defined in JSON.

The second core technological aspect of Astrakode pertains to the platform's front-end engine. This layer is built using a combination of React and Angular, and it is capable of parsing the meta-ontologies provided by the JSON files. Consequently, it dynamically renders components within the visual IDE.[Ast24]

A. Network Composer

The Network Composer is a cornerstone of the AstraKode platform, specifically designed to simplify the creation and management of blockchain networks. This tool provides a visual environment where users can configure and deploy blockchain networks without requiring deep technical expertise. It is particularly valuable for businesses that need customized blockchain solutions but lack the in-house skills to develop them from scratch.

The Network Composer allows users to visually design their blockchain networks by dragging and dropping components onto a canvas. This intuitive interface reduces the complexity of network setup, enabling users to focus on their specific requirements rather than the technical details. The tool supports the creation of various types of nodes, including permissioned and public nodes, allowing users to configure these nodes to meet their specific needs, whether for testing, development, or production environments. Once the network configuration is complete, the Network Composer automates the deployment process, which includes setting up the necessary infrastructure, installing software, and initializing the blockchain network. Such automation ensures a seamless and error-free deployment experience.

Furthermore, the Network Composer allows users to easily scale their blockchain networks by adding or removing nodes as needed. This flexibility ensures that the network can grow and adapt in response to changing business requirements. By offering an intuitive and powerful tool for network configuration and deployment, AstraKode's Network Composer lowers the barriers to blockchain adoption, enabling businesses to implement their blockchain solutions quickly and efficiently.

B. Smart Contract IDE

The Smart Contract Integrated Development Environment (IDE) is another critical component of the AstraKode platform, designed to streamline the development, testing, and deployment of smart contracts. The Smart Contract IDE provides a comprehensive suite of tools that cater to both novice and experienced developers.

The IDE offers a visual interface for designing smart contracts (figure 2), where users can utilize drag-and-drop features to create and configure contract components, reducing the need for extensive coding. This visual approach makes it easier for non-technical users to participate in the development process. To simplify the development process further, the IDE includes a library of pre-built templates for common use cases. These templates serve as starting points that users can customize to fit their specific needs, accelerating development and ensuring best practices are followed.

For users who prefer or require direct coding, the IDE includes a powerful code editor with syntax highlighting, code completion, and error checking. This editor supports multiple programming languages commonly used for smart contracts, such as Solidity. The IDE provides integrated tools for testing and debugging smart contracts, allowing users to simulate contract execution, identify and fix errors, and ensure that their contracts behave as expected before deploying them to the blockchain.

To manage changes and collaborate on smart contract development, the IDE includes version control features. Users can track changes, revert to previous versions, and collaborate with team members in a controlled and organized manner.

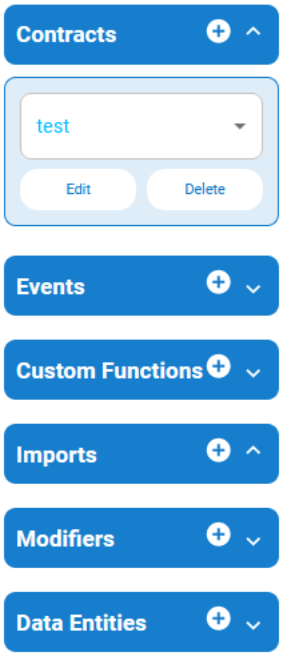

Fig. 2. Smart Contract IDE ToolBar

In the following sections, we will delve deeper into the functionalities of the Smart Contract IDE, exploring its features and capabilities in greater detail. This detailed exploration will highlight how AstraKode's tools can be used to develop robust and secure blockchain applications efficiently and effectively.

1) Contracts

In the AstraKode Smart Contract IDE, a contract serves as the fundamental unit of work for blockchain development on the Ethereum platform. Contracts are essentially classes that contain variables, functions, and structures. They encapsulate the business logic and state of a decentralized application. Each contract has a unique address on the blockchain, and it is identified by its

Application Binary Interface (ABI), which defines the functions and structures available to interact with off-chain components.

Contracts are created using a combination of code and visual tools within the IDE. The visual tools simplify the process, allowing developers to design contracts using drag-and-drop components. The IDE supports Solidity, the primary language for Ethereum smart contracts, and ensures that all contracts are compliant with Ethereum standards. This ensures compatibility and interoperability with other Ethereum-based applications.

2) Data Entities

Data entities in the AstraKode Smart Contract IDE represent the structures that hold data within a contract. These entities are analogous to classes or structs in traditional programming, used to define custom data types. They encapsulate related properties and methods, providing a clear structure for storing and manipulating data.

The IDE allows developers to define data entities visually, specifying attributes and their types through an intuitive interface. This approach simplifies the creation of complex data structures and ensures that they are correctly integrated into the smart contracts. By abstracting the complexity of data entity definition, the IDE makes it easier for developers to focus on the logic and functionality of their dApps.

3) Elements

Elements in the AstraKode Smart Contract IDE are the building blocks used to construct smart contracts. These include variables, functions, events, and modifiers, each playing a specific role in the contract's operation. Elements are defined and manipulated using the IDE's visual tools, which streamline the development process.

Variables store data, functions define the contract's behavior, events provide a mechanism for logging and notifying off-chain components, and modifiers add conditional logic to functions. By providing a comprehensive set of elements, the IDE allows developers to create robust and flexible smart contracts without deep technical knowledge of Solidity.

4) Events

Events in Ethereum smart contracts are used to log information and communicate with off-chain components. When an event is emitted, it is recorded in the blockchain's transaction log, making it accessible to external systems. This mechanism is crucial for building responsive and interactive dApps.

In the AstraKode Smart Contract IDE, events are defined visually, making it easier for developers to integrate them into their contracts. The IDE provides tools for specifying event parameters and associating them with contract functions. This visual approach simplifies the process of creating and managing events, ensuring that they are correctly implemented and utilized.

5) Functions

Functions are the core components of Ethereum smart contracts, defining the behavior and logic of the application. Each function can perform specific actions, modify the contract's state, or interact with other contracts and off-chain systems. Functions can be public, private, or have restricted access using modifiers.

The AstraKode Smart Contract IDE provides a visual interface for defining and managing functions. Developers can specify

function parameters, return types, and access modifiers using drag-and-drop tools. This visual approach reduces the complexity of function creation and ensures that they are correctly integrated into the contract's logic.

6) Imports

Imports allow developers to include external code libraries and dependencies in their smart contracts. This feature is essential for reusing code and leveraging existing solutions, such as OpenZeppelin contracts, which provide secure and tested implementations of common functionalities.

The AstraKode Smart Contract IDE simplifies the import process by providing a visual interface for adding and managing dependencies. Developers can search for and include external libraries, ensuring that their contracts are built on reliable and well-tested code. This feature enhances the modularity and maintainability of smart contracts.

7) Mapping

Mappings in Ethereum smart contracts are key-value pairs used to store and retrieve data efficiently. They are particularly useful for creating data structures like hash tables, which can quickly access information based on a unique key.

The AstraKode Smart Contract IDE provides tools for defining and managing mappings. Developers can specify the types for keys and values and use mappings to create complex data structures within their contracts. This visual approach makes it easier to implement and understand mappings, reducing the likelihood of errors and improving the contract's overall structure.

8) Modifiers

Modifiers are functions that alter the behavior of other functions in Ethereum smart contracts. They are used to enforce access control, validate inputs, and implement conditional logic. Modifiers enhance the modularity and reusability of code by allowing common logic to be applied across multiple functions.

In the AstraKode Smart Contract IDE, modifiers are defined and managed visually. Developers can create modifiers by specifying the conditions and logic they should enforce. The IDE ensures that modifiers are correctly applied to functions, improving the security and robustness of the contract.

9) Natural Language Metadata

Natural language metadata provides human-readable descriptions and documentation for smart contract components. This metadata is essential for improving the understandability and maintainability of contracts, especially in collaborative environments.

The AstraKode Smart Contract IDE allows developers to add and manage natural language metadata for all contract components. This feature ensures that contracts are well-documented, making them easier to review and audit. By integrating metadata directly into the development process, the IDE promotes best practices and enhances the overall quality of the contracts.

10) Operations

Operations in the AstraKode Smart Contract IDE refer to the actions that can be performed on contract components, such as deploying, invoking, and managing functions. The IDE provides tools for automating these operations, ensuring that they are executed correctly and efficiently.

Developers can define operations visually, specifying the parameters and conditions for

each action. The IDE automates the execution of operations, reducing the risk of errors and simplifying the management of smart contracts. This feature is crucial for ensuring that contracts are deployed and maintained effectively, supporting the ongoing functionality of the dApp.

IV. NFT MARKETPLACE

The NFT marketplace aims to provide a comprehensive and user-friendly platform for buying, selling, and refunding NFTs. The proposed marketplace is fully modular, which means that it is possible to update and redeploy only one contract without compromising the functionality of the marketplace. This modularity allows for significant savings on gas fees, as it avoids the need to redeploy the entire set of smart contracts with each update. The NFT marketplace is built on the ERC1155 protocol and includes several key features that differentiate it from other digital marketplaces.

The aim of this paper is to successfully deploy the blockchain for the NFT marketplace automatically using Katena. Given the JSON files and the respective Solidity files used to build the blockchain underlying the marketplace, our objective was to deploy it using Katena to achieve the actual deployment of the aforementioned chain [Pie24].

Main concepts of the proposed marketplace include:

1) *NFT Minting*: Creating unique digital assets on a blockchain using smart contracts.
2) *NFT Listing*: A marketplace where creators or sellers can list NFTs for other users to view and purchase.
3) *NFT Sale*: The ability to purchase an NFT in a single transaction.
4) *Fee Manager*: A fully configurable contract that allows clients to set fees and fee receiver addresses.
5) *Encoding Metadata*: The ability to encode metadata in the NFT by passing it to the minting function.
6) *Refund Process*: A refunding procedure that uses a configurable Chainlink Oracle.
7) *Compatibility*: The code is compatible with any EVM, including Ethereum, Polygon, Avalanche, and more.

A. Main Contract

The main contract is used as a hub to spread function calls to other contracts, such that the UI primarily interacts with the main contract to ensure the functionality of the entire marketplace [Pie24].

B. NFT Contract

The NFT contract is a smart contract that implements the ERC1155 standard for non-fungible tokens (NFTs) [Pie24].

C. MintFactory

The MintFactory contract is a smart contract that allows the creation of new NFT-Contract instances [Pie24].

D. MarketPlace

The MarketPlaceMain1155 contract is a smart contract that implements a marketplace for selling and buying items. The items in the marketplace can be represented by non-fungible tokens (NFTs) that conform to the IERC1155 interface [Pie24].

E. FeeManager

The FeeManager contract is a smart contract that allows an operator to set a fee and

a fee receiver, and to transfer funds from one address to another with a specified fee [Pie24].

F. GetOracleValue

The GetOracleValue contract is a smart contract that allows users to retrieve the latest volume data for a specific region from an external API and store it on the Ethereum blockchain. The contract makes use of the ChainlinkClient and Ownable contracts and utilizes the Chainlink library [Pie24].

G. Refunded

The Refunded contract is a smart contract that allows for the refunding of a purchase to buyers under certain circumstances [Pie24].

V. Implementation with Katena

A. Code Implementation

Developing a distributed application requires a significant amount of effort. As mentioned earlier, blockchain technology is still evolving. Its mechanisms and inherently independent nature, free from third-party intermediaries, demand a deep understanding of the underlying concepts. For example, when we have built the NFT MarketPlace model, the logical steps to follow included:

1) **Designing the software model** that incorporates all the necessary functionalities, using an appropriate development methodology such as SCRUM or AGILE.
2) **Writing the code for each smart contract** defined during the design phase. This process is facilitated by the AstraKode Tool, which enables users to write complex code through an intuitive graphical interface.
3) **Compiling the smart contracts** using advanced frameworks like Truffle or HardHat. At this stage, since the EVM does not execute high-level languages like Solidity directly, each contract is translated into bytecode, the format compatible with the virtual machine. This phase also generates the ABI, which allows interaction with the contract once it is deployed on the blockchain.
4) **Writing the deployment scripts for the contracts.** During this stage, the compiled bytecode is sent to the blockchain network via a transaction. Once confirmed, the contract becomes active and is assigned a unique address. From this point onward, it is accessible and usable by anyone on the network, highlighting the importance of ensuring that the contract is free of vulnerabilities. Any errors or security flaws present after deployment could be exploited by malicious actors, with potentially disastrous consequences. A critical aspect of this process is the order in which these contracts are developed, as they need to interact with each other seamlessly.

The final stage is managed by the Katena framework, which allows the entire functionality of the MarketPlace to be described in a single YAML file. The user only needs to declare the nodes (and their properties) that interact within the distributed app, without worrying about how these interactions occur or the dependencies between them. This approach automates and secures most of the operations that are error-prone. In our case, we developed the entire topology using **Ganache** for local testing. Ganache is a per-

sonal blockchain for Ethereum development that can be used to test smart contracts and DApps in a sandbox environment. It can simulate various network conditions, such as latency and limited bandwidth, allowing for comprehensive performance testing of smart contracts and DApps [Blo24]. Let us now demonstrate the results using Katena:

```
tosca_definitions_version:
    tosca_simple_yaml_1_3

imports:
- nodes/contract.yaml
- nodes/network.yaml
- nodes/wallet.yaml

topology_template:
  node_templates:
    ganache:
      type: katena.nodes.network.ganache
    userWallet:
      type: katena.nodes.wallet
      requirements:
      - host: ganache
      properties:
        privateKey: { get_input:
            UserKeyGanache }
        owner: { get_input: UserWallet}

    ##### LIBRARIES #####

    counters:
      type: katena.nodes.library
      requirements:
      - host: ganache
      - wallet: userWallet
      properties:
        abi: "Counters"

    marketItemData:
      type: katena.nodes.library
      requirements:
      - host: ganache
      - wallet: userWallet
      properties:
        abi: "MarketItemData"

    refundedData:
      type: katena.nodes.library
      requirements:
      - host: ganache
      - wallet: userWallet
      properties:
        abi: "RefundedData"

    contractCreated:
      type: katena.nodes.library
      requirements:
      - host: ganache
      - wallet: userWallet
      properties:
        abi: "ContractCreated"

    ##### CONTRACTS #####

    marketplace:
      type: katena.nodes.contract
      requirements:
      - host: ganache
      - wallet: userWallet
      properties:
        abi: "MarketPlace"
        parameters:
          - 10
          - 50

    refunded:
      type: katena.nodes.contract
      requirements:
      - host: ganache
      - wallet: userWallet
      properties:
        abi: "Refunded"
        parameters:
          - 10
          - 50
          - 0xb0897686c545045aFc77CF20eC
          7A532E3120E0F1

    nftContract:
      type: katena.nodes.contract
      requirements:
      - host: ganache
      - wallet: userWallet
      properties:
        abi: "NFTContract"
        parameters:
          - https://www.google.com/
          - [1]
          - [1]
          - userWallet # First account
              taken by Web3

    mintFactory:
      type: katena.nodes.contract
      requirements:
      - host: ganache
      - wallet: userWallet
      properties:
        abi: "MintFactory"
        parameters:
        - userWallet # First account
            taken by Web3
        - false # RoleEnable

    main:
      type: katena.nodes.contract
      requirements:
```

```
      - host: ganache
      - wallet: userWallet
      - constructorCalls: marketplace
      - constructorCalls: refunded
      - constructorCalls: mintFactory
      properties:
        abi: "Main"

  inputs:
    UserKeyGanache:
      type: string
      required: true
    UserWallet:
      type: string
      required: true
```

If we had wanted to develop this same topology using a state-of-the-art framework like Truffle, we would have obtained the following deployment script:

```
const Counters = artifacts.require("Counters");
const MarketItemData = artifacts.require("MarketItemData");
const RefundedData = artifacts.require("RefundedData");
const Main = artifacts.require("Main");

const ContractCreated = artifacts.require("ContractCreated");
const MarketPlace = artifacts.require("MarketPlace");
const GetOracleValue = artifacts.require("GetOracleValue");
const MintFactory = artifacts.require("MintFactory");
const Refunded = artifacts.require("Refunded");
const NFTContract = artifacts.require("NFTContract");
const FeeManager = artifacts.require("FeeManager");
const linkAddress = "0xb0897686c545045aFc77CF20eC7A532E3120E0F1";
const Web3 = require("web3");


web3 = new Web3(web3.currentProvider);
let accounts;

//flag for enabling/disabling refunds
const isRefundEnabled = true;
const RoleEnable = false;

module.exports = async function (deployer) {
  //retrieves truffle accounts list
  // instead of hardcoding
  try {
    accounts = await web3.eth.getAccounts();
    console.log(accounts);
  } catch (error) {
    console.error(error);
  }
  console.log(accounts);
  await deployer.deploy(Counters);

  await deployer.deploy(MarketItemData);
  await deployer.deploy(ContractCreated);

  await deployer.link(Counters, MarketPlace);
  await deployer.link(MarketItemData, MarketPlace);
  await deployer.deploy(MarketPlace, 10, 50);
  if (isRefundEnabled) {
    await deployer.deploy(RefundedData);

    await deployer.link(Counters, Refunded);
    await deployer.link(RefundedData, Refunded);
    await deployer.deploy(Refunded, 10, 50, linkAddress);
    console.log("Refunded: " + Refunded.address);
    let refundedAdress = Refunded.address;
    console.log("Refunds Enabled");
  } else {
    console.log("Refunds Disabled");
  }

  await deployer.link(Counters, MintFactory);
  await deployer.link(ContractCreated, MintFactory);
  await deployer.deploy(MintFactory, accounts[0], RoleEnable);

  await deployer.deploy(
    NFTContract,
    "https://www.google.com/",
    [1],
    [1],

    accounts[0]
  );

  await deployer.link(MarketItemData, Main);
};
const Main = artifacts.require("Main");

const MarketPlace = artifacts.require("
```

```
    ./MarketPlace.sol");
const Refunded = artifacts.require("
    ./Refunded.sol");
const MintFactory = artifacts.require("
    ./MintFactory.sol");
const isRefundEnabled = false;
module.exports = async function (deployer
    ) {
  await deployer.deploy(
    Main,
    MarketPlace.address,
    Refunded.address,
    MintFactory.address
  );
};
```

B. Comparison Between The Two Approaches

Both code snippets deploy smart contracts on a blockchain, but they use two distinct approaches. While both aim to achieve the same goal, the Katena approach offers several advantages that make it superior.

1) Abstraction and Modularity

Katena takes a declarative approach, where the infrastructure and smart contract components are defined using YAML, a format that is readable by both humans and machines. This abstract approach allows for clear descriptions of dependencies between components and modular management of configurations. For example, node definitions like Ganache, userWallet, and various contracts are all separated, with their relationships explicitly defined through requirements. This improves code comprehension and maintenance, as each component can be easily updated or replaced without affecting the rest of the configuration. In contrast, Truffle code is more imperative and linear, requiring the writing of functions for each deployment phase. While powerful, it is less flexible and more prone to human error during code updates or modifications. Dependencies and interactions between contracts are not clearly articulated, which can make managing complex projects more difficult.

2) Reusability and Automation

Katena enhances reusability through its modular structure. Components like libraries and contracts can be defined once and easily reused in different contexts. Additionally, Katena supports the automation of orchestration and deployment operations, making it simpler to scale infrastructure or perform complex deployments across different environments. This is particularly useful in development and production contexts where the same configurations can be used with minimal modifications. Although Truffle is powerful, it does not offer the same level of reusability. Each migration script must be written and maintained manually, increasing complexity when managing multiple contracts or working across different networks. While Truffle is an excellent tool for smaller development environments or single projects, it becomes less efficient in contexts that require high levels of automation and infrastructure management.

3) Scalability and Maintenance

Katena is designed to handle complex applications and infrastructures, allowing for scalable management of blockchain resources. The ability to define topological templates in YAML enables easy management of multi-node and multi-contract environments. Maintenance is simplified, as each component can be updated or redeployed without affecting the entire network. In contrast, Truffle's model can become cumbersome and difficult to maintain as project complexity increases. Managing multiple contracts and integrating new features

require significant changes to existing migration scripts, increasing the risk of errors and necessitating greater attention during development and deployment.

4) Interoperability and Standardization

The use of TOSCA with Katena introduces a level of standardization that facilitates interoperability across different platforms and technologies. This is especially important in enterprise environments where solutions must integrate with existing or future infrastructures. Katena, based on open standards, provides a common language that can be understood and implemented across various cloud platforms, ensuring greater portability of blockchain applications. In contrast, Truffle is specific to the Ethereum ecosystem and does not offer the same level of interoperability. While it is a powerful framework within its domain, it is not designed to manage complex infrastructures that may involve multiple technologies or platforms.

VI. Conclusions and future work

The integration of low-code platforms like AstraKode with declarative frameworks such as Katena represents a significant advancement in simplifying blockchain development. This combination not only democratizes access to blockchain technology but also bridges the gap between high-level design and the intricacies of blockchain implementation. The NFT Marketplace project discussed in this paper serves as a compelling example of how these tools can be leveraged to streamline the development process and reduce the technical barriers traditionally associated with blockchain.

However, the current implementation of the AstraKode Tool, even if promising, still it requires further development to fully realize its potential in more complex scenarios, such as the complete deployment and management of an NFT Marketplace. Despite this, the intuitive visual interface and the low-code approach offered by AstraKode have already shown significant potential in enabling users with limited technical expertise to create and deploy blockchain applications.

Looking forward, a promising direction for future work would involve the seamless integration of AstraKode's low-code smart contract creation tool with Katena's declarative approach for deployment and management. By automating the entire process—from the writing of smart contracts to their deployment and lifecycle management—developers could significantly enhance efficiency and reduce the potential for errors. This integration could also facilitate a more cohesive development experience, where users can design, deploy, and manage their blockchain applications within a unified platform, ultimately making advanced blockchain technology more accessible to a broader audience.

VII. ACKNOWLEDGEMENTS

Davide Gigante and Federica Topazio have been supported by Fabiano Izzo (founder of AstraKode) and Alessandro Sommaruga, while studying and testing AstraKode Tool. Special thanks go to Damian Tamburri and Giovanni Quattrocchi from Politecnico di Milano for their academic guidance and constructive feedback throughout the project.

References

[BCW12] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-Driven Software Engineering in*

Practice. Vol. 1. Synthesis Lectures on Software Engineering. Morgan & Claypool, 2012.

[FV15] Vogelsteller Fabian and Buterin Vitalik. "Eip-20: Erc-20 token standard". In: *Ethereum Improvement Proposals* 20 (2015).

[Hil+17] Everett Hildenbrandt et al. "Kevm: A complete semantics of the ethereum virtual machine". In: (2017).

[Sta+17] M Staples et al. "Risks and opportunities for systems using blockchain and smart contracts. Data61". In: *CSIRO), Sydney* (2017).

[Che18] Sylve Chevet. "Blockchain technology and non-fungible tokens: Reshaping value chains in creative industries". In: *Available at SSRN 3212662* (2018).

[Li+18] Dongxing Li et al. "A blockchain-based authentication and security mechanism for IoT". In: *2018 27th International Conference on Computer Communication and Networks (ICCCN)*. IEEE. 2018, pp. 1–6.

[Moh18] Debajani Mohanty. "Ethereum for architects and developers". In: *Apress Media LLC, California* (2018), pp. 14–15.

[Zhe+18] Zibin Zheng et al. "Blockchain challenges and opportunities: A survey". In: *International journal of web and grid services* 14.4 (2018), pp. 352–375.

[NR19] Nawari O Nawari and Shriraam Ravindran. "Blockchain and the built environment: Potentials and limitations". In: *Journal of Building Engineering* 25 (2019), p. 100832.

[Bra+20] Nils Braun-Dubler et al. *Blockchain: Capabilities, economic viability, and the socio-technical environment*. Vol. 73. vdf Hochschulverlag AG, 2020.

[CB20] Yan Chen and Cristiano Bellavitis. "Blockchain disruption and decentralized finance: The rise of decentralized business models". In: *Journal of Business Venturing Insights* 13 (2020), e00151.

[Has+20] Vikas Hassija et al. "A survey on supply chain security: Application areas, security threats, and solution architectures". In: *IEEE Internet of Things Journal* 8.8 (2020), pp. 6222–6246.

[MSS20] Koushik Bhargav Muthe, Khushboo Sharma, and Karthik Epperla Nagendra Sri. "A Blockchain Based Decentralized Computing And NFT Infrastructure For Game Networks". In: *2020 Second International Conference on Blockchain Computing and Applications (BCCA)*. 2020, pp. 73–77. DOI: 10 . 1109 / BCCA50787.2020.9274085.

[PPP20] Kyleen W Prewett, Gregory L Prescott, and Kirk Phillips. "Blockchain adoption is inevitable—Barriers and risks remain". In: *Journal of Corporate accounting & finance* 31.2 (2020), pp. 21–28.

[San+21] Abdurrashid Ibrahim Sanka et al. "A survey of breakthrough in blockchain technology: Adoptions, applications, challenges and future research". In: *Computer communications* 169 (2021), pp. 179–201.

[Wan+21] Qin Wang et al. "Non-fungible token (NFT): Overview, evaluation, opportunities and challenges". In: *arXiv preprint arXiv:2105.07447* (2021).

[Bar+22] Luciano Baresi et al. "A declarative modelling framework for the deployment and management of blockchain applications". In: *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems*. 2022, pp. 311–321.

[Jav+22] Mohd Javaid et al. "A review of Blockchain Technology applications for financial services". In: *BenchCouncil Transactions on Benchmarks, Standards and Evaluations* 2.3 (2022), p. 100073. ISSN: 2772-4859. DOI: https://doi.org/10.1016/j.tbench.2022.100073. URL: https://www.sciencedirect.com/science/article/pii/S2772485922000606.

[CHF23] Simon Curty, Felix Härer, and Hans-Georg Fill. "Design of blockchain-based applications using model-driven engineering and low-code/no-code platforms: a structured literature review". In: *Software and Systems Modeling* 22.6 (2023), pp. 1857–1895.

[Hay23] Adam Hayes. *Non-Fungible Token (NFT): What It Means and How It Works*. 2023. URL: https://www.investopedia.com/non-fungible-tokens-nft-5115211 (visited on 06/26/2024).

[Nav+23] Nigam Naveed et al. "Efficient, Immutable and Privacy Preserving E-Healthcare Systems using Blockchain". In: *2023 International Conference on Communication Technologies (ComTech)*. 2023, pp. 140–145. DOI: 10.1109/ComTech57708.2023.10164855.

[Ast24] Astrakode. *Astrakode Wiki*. 2024. URL: https://docs.astrakode.tech/akb-wiki/astrakode-wiki (visited on 06/26/2024).

[Blo24] Blockchain Council. *Ganache Blockchain*. 2024. URL: https://www.blockchain-council.org/blockchain/ganache-blockchain-all-you-need-to-know/.

[Pie24] PieceX. *Full NFT Marketplace Smart Contracts integrated with customisable oracle*. 2024. URL: https://www.piecex.com/source-code/Full-NFT-Marketplace-Smart-Contracts-integrated-with-customisable-oracle-5723 (visited on 06/26/2024).