IEEE Open Journal of
Systems Engineering

Received 31 August 2022; revised 20 December 2022; accepted 28 March 2023. Date of publication 31 March 2023;
date of current version 28 April 2023.

Digital Object Identifier 10.1109/OJSE.2023.3263504

ChaordicLedger: Knowledge Transfer for Industry

MICHAEL F. MARCHINI [1,2]

[1]Lockheed Martin Space, Denver, CO 80201 USA
[2]College of Electrical and Computer Engineering, Villanova University, Villanova, PA 19085 USA

This work was made possible by research and development funding by Lockheed Martin Space.

ABSTRACT In the development of large-scale, integrated systems, consistent clarity and distribution of knowledge and relevant status are crucial to success. There are many existing methods and commercial tools for providing context and traceability from requirements and specifications to a system's artifacts, but these tools are often vendor locked, require access to cloud-based services, and necessitate high licensing costs. Furthermore, modern large-scale system development involves multiple business partners, each of which needs to ensure their teams have granular, role-based access to all relevant information without impediment; centralized warehousing and gatekeeping should not be handled by a single entity if the information is to remain readily accessible. Instead, a permissioned, distributed knowledgebase that avoids vendor lock-in enables a consistent, real-time view of information that provides equivalent context for developers and other stakeholders. ChaordicLedger, a free and open-source project, joins the transparency and smart contract aspects of distributed ledger technology with the storage capabilities of a distributed file system to fulfill this industrial application while allowing for industry-specific customizations.

INDEX TERMS Business intelligence, digital transformation, distributed ledger, software development, systems engineering.

I. INTRODUCTION

Knowledge transfer is crucial to the development of large-scale, integrated systems, especially considering the vast and disparate legal and contractual obligations stipulated by a system's purpose and environment. Once a system is created and deemed operational, there exists a period of maintenance until the system is decommissioned; during this period, personnel must remain informed of necessary changes in response to mission needs or environmental perturbation. Often, high-level changes to system interfaces are announced through an engineering change proposal, and the maintainers of systems implementing the interface will take that information, perform an analysis, and return with an estimated time of completion and a cost impact. This process can be straightforward and accurate if all personnel supporting the system have equitable insights and knowledge of the system (e.g., it has the original development team, and the system was recently deployed). However, the process is arduous and inaccurate if the team lacks access to subject–matter experts (SMEs), if the original development team is inaccessible, or if the supporting team is composed of newer personnel who lack implicit and deep familiarity with the system; over time, especially for companies supporting multiple systems, this latter case is typical.

A company may have its own proprietary tool for this purpose, or a team may use a commercial tool like International Business Machines (IBM) Rational Dynamic Object-Oriented Requirements System (DOORS) [1]. In the former case, a proprietary tool requires internal maintenance and development, whereas in the latter case, the company is subject to costly licensing and maintenance agreements and dependency on a specific vendor (known as a vendor lock-in). Furthermore, these tools necessitate the use of centralized servers or cloud-based vendor-provided services that restrict collaboration between business partners and limit (or render impossible) portability across security domains from open to air-gapped systems. Parallel to the issue of deciding what type of traceability system to use is the decision of how to manage the artifacts that verify the requirements in the requirement verification traceability matrix [2].

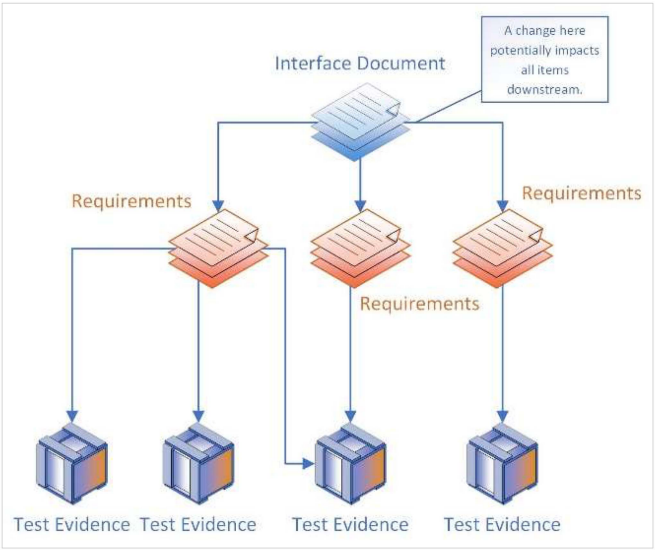

FIGURE 1. Example of interrelated artifacts [3].

To ease this burden and allow change impact analyses to minimize subjectivity and maximize accuracy, disciplined knowledge transfer and diffusion across all personnel (both new and tenured) is required, which is a cultural difficulty in fast-paced teams and teams that support legacy systems; tasking teams to create vast knowledgebases is an expensive proposition, as is making everyone an expert on every system. Instead, tooling can be used to demonstrate the connectivity between critical documents, their explicit and derived requirements, and the captured artifacts that demonstrate implementation of each related specification. To be useful in an industrial enterprise context, the tooling must support the ability to add artifacts and relationships at any time (while abiding by appropriate business rules and access controls), and activity within the tooling must be auditable to ensure data provenance. Furthermore, the tooling should enable anyone with any level of experience to start with a specific artifact and visualize every related artifact that would be impacted by a change (see Fig. 1). A solution exists in ChaordicLedger, a free and open-source software product created for this purpose. This tool fulfills the industrial need for a consistent, real-time view of information, providing equivalent context for developers and other stakeholders without the overhead of license costs and vendor lock-in. Furthermore, the transparency of contextualized information reduces or eliminates the dependencies on SMEs to assess systemic change impact, allowing novices to provide reliable estimates equivalent to those of their more experienced counterparts.

II. BACKGROUND

The field of complex systems engineering covers the creation of systems with multitudes of dependencies and interlocking parts that must work together in unison to achieve a desired outcome and be reliable over time. Complex systems engineering has an often-overlooked aspect: the processes that enable it; as dynamic as the system and its requirements are over time, so are the corresponding methodologies and development personnel for implementing the system [4]. While working within applicable legal and moral bounds, people are unpredictable and may execute their work in surprising and inspired ways to meet their objectives; this is especially true if an Agile framework is applied and teams are given the correct tools to perform their work, left alone, and trusted to deliver a quality product [5], [6]. Complex systems may "operate in the "chaordic" region between chaos and order. The healthier and more effective CSs operate at the "edge of chaos." They rarely remain in any given state for long but may have transient and relatively fleeting states of stability" [7]. The need to bring just enough order to the chaos is where ChaordicLedger gets its name and purpose.

Beyond support for complex systems engineering, a tangible impact exists in the ability to predict costs, estimate change impact, and quantify value. "Software development methods that integrate requirements analysis with the organizational and external requirements are crucial in view of the fact that socio-technical systems are a vital part of the organization's value chain" [8]. Using a tool to enable linkage between artifacts has the additional benefit of providing unambiguous inputs to the cost model for verifying requirements (CMVR), which is predicated on the traceability of requirements from stakeholders down to individual tests [9]. The CMVR describes the cost–scope–time pyramid, with each impacting the other; specifically, changes to requirements are represented as changes to the "scope" aspect, which directly impacts the other aspects.

Once a complex system is developed and deployed, it moves into an operations and maintenance phase, and if the system is operational for more than a few years, it can be considered a "legacy" system. Codebases for legacy systems leverage computations (known as "actuals") as applied to the constructive cost model (COCOMO) II [10], which has numerous scale factors beyond code count, including development flexibility, team cohesion, process maturity, and the degree to which something is precedented. Unfortunately, the scale factors do not directly factor in the accessibility of relevant artifacts, such as interface control documents and test evidence. With ChaordicLedger, COCOMO II predictions can use more efficient tunings for scale factors. This results in both a reduction in the cost to develop or modify systems and an increase in confidence for new business acquisitions.

When basing decisions on data (e.g., computing value and estimating impact, etc.), objectivity is established on the reliability and traceability of the data sources; if the data are not trustworthy, the decisions based on them will be unreliable. Therefore, the origins of data must be assured in any methodology or system that provides access to data and metadata. Typically, such provenance is assured through a configuration management (CM) organization, which is a special group tasked with manually ensuring that only authorized individuals can access, provide, revise, review, and approve critical artifacts for development efforts. While this

process provides provenance for the captured artifacts, the amount of procedural overhead discourages frequent updates to the artifacts, and, as such, the artifacts become simply a snapshot of a point in time at which a verification occurred. This procedural latency is magnified when modern software development approaches, like development, security, and operations (DevSecOps) [11], [12], are considered, wherein a go-fast mentality is applied while remaining conscious of quality, security, and the needs of the end user. Within DevSecOps, multiple new versions of a product may be produced daily or hourly, each with a complete software bill of materials (to list its parts/dependencies) augmented by static code analysis results and run-time vulnerability assessments; all of these artifacts of system development can be emitted by a DevSecOps pipeline, stored within ChaordicLedger, and interrelated to enable traceability to requirements and specifications.

A study of "three large ISO-9001 certified, high process maturity firms that were also assessed at the CMMI Level-5" [13] found that software project estimation via "regression-based methods failed due to three main reasons": 1) not accounting for intrinsic differences in globally distributed software project configuration; 2) not having inputs for comprehensive metrics at the start of a project; and 3) regression methods were very sensitive to the experience level of the project manager using them. A solution to these issues is in the quality and transparency of available project-related data, which allows for objective differentiation between projects (issue 1), mining of relevant metrics for cost models (issue 2), and mitigating the need for subjective expert insights (issue 3) [3].

The business rules enforced and executed by a CM organization can be codified into a transaction between a user and a traceability system in the form of a smart contract, which is a software-defined, Turing-complete method of creating executable business rules embedded in the traceability system; these rules mitigate the human-in-the-loop vector for error introduction and enable novice programmers to define and maintain the rules per shifting business needs. Smart contracts are a first-class feature of various distributed ledger technology (DLT) implementations, such as Hyperledger Fabric, wherein smart contracts are referred to as "chaincode" [14].

The primary drivers for developing ChaordicLedger are to address these industrial software development needs and to discover the validity of the assertion that "Interplanetary File System (IPFS) and the Blockchain are a perfect match" [15] and whether IPFS (an implementation of a distributed file system, or DFS) and blockchain (an implementation of DLT) can effectively be combined into a unified system [16]. The combination of DLT and DFS, as formerly separate technologies with distinct purposes, builds upon the state-of-the-art by creating an approachable system that provides a macro-to-micro scale view of large integrated systems in support of systems engineering, CM, software engineering, and cost account management without the risk of inflexible customizations or vendor-lock. A commercial product called ThunderChain File System (TCFS) implements a similar concept of combining a blockchain with IPFS, but it is built upon Ethereum, requires buy-in and incentives for transaction processing, and requires persistent access to TCFS services [17]. This type of vendor-lock is counter to the objectives of ChaordicLedger, which is designed to be free and portable to multitudes of deployments, including isolated and air-gapped systems, particularly those used for processing sensitive data.

The goal of this codebase is to establish a technology readiness level (TRL) 4 [18] proof of concept (PoC) of integrating the distributed nature of IPFS with the permissioned, private nature of Hyperledger Fabric via industry-relevant smart contracts to achieve a macro-to-micro scale view of large integrated systems in support of systems engineering, CM, software engineering, cost account management, and any other stakeholder; its purpose is to provide a consistent, on-demand worldview to mitigate or eliminate the need to use subjective "engineering judgment" in favor of using objective actuals in change impact analysis. Furthermore, the use of this PoC would not incur the costs of cloud-based or cloud-backed solutions (such as deployments of IBM DOORS).

It is important to note that ChaordicLedger is more applicable to rapid application development and go-fast initiatives that may be implemented as nonlife-critical software development via Agile Methodologies, wherein requirements may change every two or three weeks. This does not preclude its use in the blended complex systems (CS) development of both software and hardware solutions, if anything it magnifies the importance and criticality of well-defined interfaces and the traceability of test evidence.

III. LEGACY TOOLS AND TECHNOLOGIES

Modern businesses and system development cannot be assumed to exist in colocated offices under the banner of a single vendor; instead, development has evolved to include distributed teams across multiple vendors, time zones, and countries. Even with the availability of cloud providers with 24×7 accessibility worldwide, a global, centrally controlled database is insufficient to support a model of both Zero Trust and distributed data ownership and stewardship. As such, could a technology like DLT provide a solution? Its use must pass the rubric of applicability shown in Table 1.

Since DLT is applicable to the domain, it must be compared against other available tools to form a valid, justifiable trade between them. Table 2 lists a set of desirable traits for tracking and sharing contextualized knowledge across disparate teams and stakeholders while enforcing relevant business rules and allowing for reliable auditability. Combining DLT and DFS yields a tool that exhibits the traits of each.

Leveraging all desirable traits of DLT and DFS addresses the findings presented in [13] by providing traceability and provenance of changes and by enabling a consistent view of the dependency space between all contextualized artifacts for developers and executives alike; such transparency enables daily, localized optimizations by development teams while enabling wide-reaching optimizations and predictability

TABLE 1. Applicability Rubric for DLT in This Domain

DLT Applicability Question	Consideration for Large-Scale, Integrated Development
Is it necessary to have a shared database?	For large-scale project management, implementation, verification, and validation, a database is necessary to track all generated artifacts. Further, since these projects must be orchestrated across numerous teams, organizations, and business partners, the database must be shared in such a manner to provide a consistent and complete view to all parties.
Is it necessary to have multiple parties writing data?	Modifications to tracked artifacts are essential to the implementation of a project, especially considering the potential for changes to overarching specifications and requirements during development.
Are potential writers untrusted (should writers be prevented from modifying others' previous entries)?	A common adage in Agile development is to empower the teams by giving them the tools to do their work and to trusting them to get the job done. Caution is exercised by applying the principle of least-privilege, and no user would have far-reaching capability to make irresponsible and damaging changes. If negative actions do occur (e.g., due to a set of malicious insiders), the actions are easily remediated by other ledger participants.
Is disintermediation needed (is it necessary to remove trusted intermediaries verifying or authenticating transactions)?	Disintermediation applies differently based on the use case. For instance, intermediaries would be favorable regarding modifications to system-level specification. However, they would be unnecessary for continually updated verification artifacts, such as results generated from automated tests.
Is it necessary to see how transactions are linked to each other (should different actors independently write transactions concerning a single user)?	In an enterprise setting, especially those bound by laws regarding auditability and change traceability, the ordering and linkage of transactions is crucial to fulfilling business objectives and the burden of rigorous scientific and engineering practices, especially those detailed in the IEEE Standard 1012-2016, "IEEE Standard for System, Software, and Hardware Verification and Validation" [19].

TABLE 2. Trait Comparison of Available Technologies [4]

Trait	Database	DLT	DFS	DLT + DFS
Transactional Control	✓	✓	✗	✓
Flexible for storing content	✓	✗	✓	✓
Permissioned Access	✓	✓	✗	✓
Relationship between Content	✓	✗	✓	✓
Append-only, shared log file	✓	✓	✗	✓
Sequential, time-stamped transactions	✓	✓	✗	✓
Auditable and tamper-proof	✗	✓	✗	✓
Decentralized / distributed across organizations	✗	✓	✓	✓
No licensing costs	✗	✓	✓	✓
Traceable transaction history	✓	✓	✗	✓
Executable business rules.	✗	✓	✗	✓
Explicit, controlled changes to content.	✗	✓	✗	✓
Applicable to the problem domain	✗	✓	✓	✓

for management. In the midst of rapidly changing requirements and their fulfilling test evidence artifacts, the usefulness of automatic business rule enforcement shines. Furthermore, ChaordicLedger may be integrated directly with continuous integration/continuous delivery pipelines, aligning with modern software development best-practices.

IV. CHAORDICLEDGER AND ITS TECHNOLOGIES

ChaordicLedger combines the capabilities of DLT and DFSs to enable a distributed and permissioned storage solution for

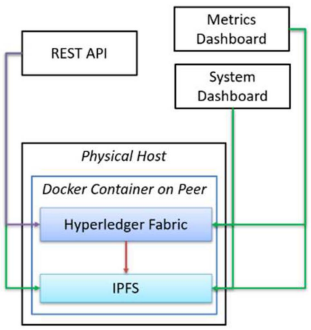

FIGURE 2. Block diagram of the application's components.

interrelated artifacts. The application is designed to execute on minimal system requirements (e.g., 4 CPU cores, 8-GB RAM) and run entirely on FOSS products, including Docker, Hyperledger Fabric, IPFS, and Kubernetes. The application supports smart contracts written in Go or Java and features a representational state transfer application programming interface (REST API) for fast, lightweight interactions via HyperText Transfer Protocol (HTTP) requests transporting JavaScript Object Notation (JSON) payloads. The application also includes microservice-level observability in the form of Elasticsearch and Kibana, wherein data are provided by Fluentd data collection. The online documentation in the application's source code repository (https://github.com/lmco/ChaordicLedger) describes the necessary prerequisites for installation (e.g., Docker). Fig. 2 shows ChaordicLedger's high-level components; read-only requests may be made to either the distributed ledger (implemented as Hyperledger Fabric) or to the DFS (implemented as IPFS), but write requests may only be made by way of the API and the smart contracts installed in the distributed ledger.

V. TECHNOLOGY READINESS

ChaordicLedger was developed as a PoC in pursuit of the author's Ph.D. research objective of determining whether DLT and DFS could be combined into a form applicable to industrial large-scale system development and integration. The application achieves TRL 4, indicating it has received component-level verification in a laboratory environment [20]. Note that the default installation uses a generated certificate authority (CA) and generated public key infrastructure (PKI) certificates for the distributed ledger's internal orders and peers, though in an enterprise setting, an enterprise root CA would generate the certificates for the ChaordicLedger's users. Furthermore, the default installation includes smart contracts that govern the creation of artifacts and their relationships but does not prevent bad actors from curtailing the API and corrupting the system's state. In its current state, ChaordicLedger can handle thousands of interrelated artifacts so long as their individual sizes do not exceed 25 Kibibytes (25 600 byte). (Note: 1 Kibibyte—KiB—is 1024 bytes; the power of 2 sizing is used by Linux systems.)

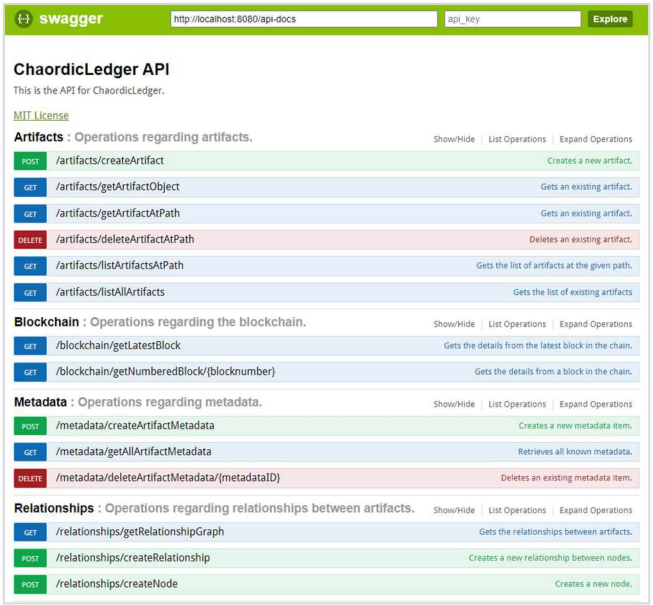

FIGURE 3. Swagger view of the REST API's methods [21].

To raise the TRL of this platform, a few technical aspects must be addressed: First is the enablement of PKI certificates for each peer node as provided by a CA, which establishes the nodes' identities. Next is the enablement of role-based authorization, which confirms that an authentic user has the authority to perform various actions. Then, the enablement of transport layer security for smart contracts. Finally, the deployment model for the platform should be tuned to avoid privileged shell access within deployed Kubernetes pods; the combination of these improvements would mitigate state corruption via bad actors. The limit of 25 KiB can be mitigated through an alternate method of uploading and downloading content by way of the REST API and the underlying interchange between the API's implementation in NodeJS, Hyperledger Fabric, and IPFS. An evolving list of future development goals is published on GitHub at https://github.com/lmco/ChaordicLedger.

VI. USING CHAORDICLEDGER

Upon installation, ChaordicLedger is in an initial, empty state containing zero artifacts and, necessarily, zero artifact relationships. This is evidenced by querying the application via its REST API (see Figs. 3–5).

The simplest test case for the application is to create a random, binary artifact and relate it to itself (while a self-referential artifact has dubious meaning, it is sufficient to illustrate the tool's capability). The following Linux Shell code snippet creates a random file of size 1 KiB and uses two uniform resource locators (URLs) to upload the file to ChaordicLedger and relate the file to itself

With the artifact uploaded and related to itself, a third API method is used to extract the graph of relationship information

```
# Define the URL for the ‘createArtifact’ API method.
url="http://localhost:8080/v1/artifacts/createArtifact"

# Capture a timestamp in ISO-8601 format.
now=`date -u +"%Y%m%dT%H%M%SZ"`

# Name the file
randomFile=randomArtifact1_${now}.bin

# Generate the file’s random contents.
head -c 1KiB /dev/urandom > $randomFile

# Execute the createArtifact method via HTTP POST and
# capture its result.
RESULT=$(curl -s -X POST -F "upfile=@${randomFile}" --header
'Content-Type: multipart/form-data' --header 'Accept:
application/json' "${url}")

# Extract the IPFS name from the result.
ipfsName=$(echo $RESULT | jq .result.result.IPFSName | tr -d '"')

# Define the URL for the ‘createRelationship’ API method.
url="http://localhost:8080/v1/relationships/createRelationship"

# Define the payload for the request.
data='{ "nodeida": "'${ipfsName}'", "nodeidb": "'${ipfsName}'"}'

# Execute the createRelationship method.
curl -s -X POST --header 'Content-Type: application/json' --
header 'Accept: application/json' -d "${data}" "${url}"
```

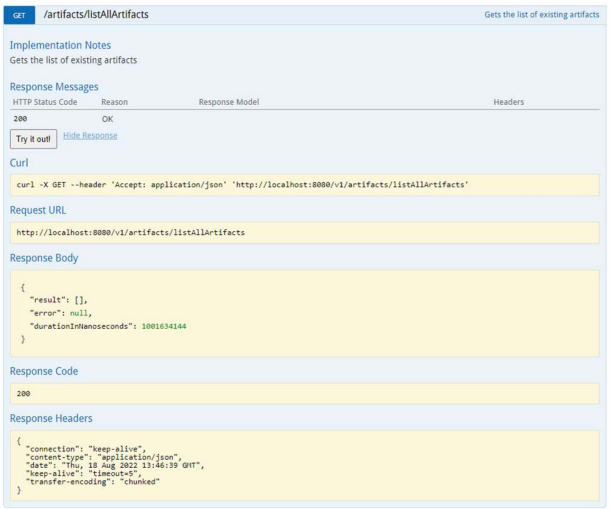

FIGURE 4. Execution of “listAllArtifacts” API method in the initial application state.

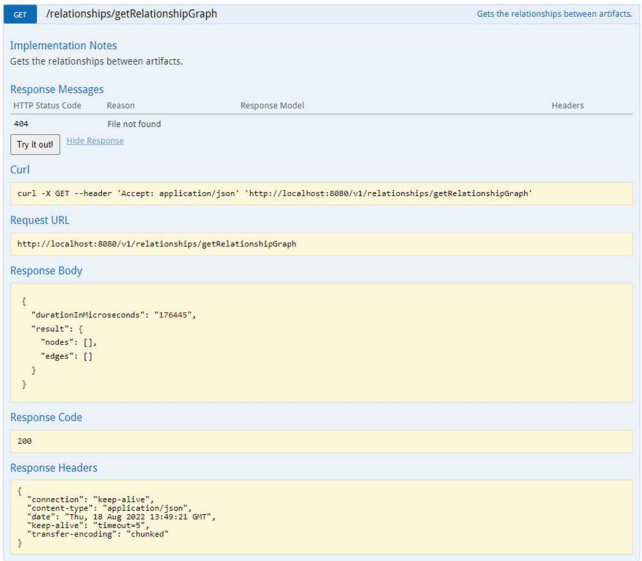

FIGURE 5. Execution of “getRelationshipGraph” API method in the initial application state.

```
# Define the URL of the getRelationshipGraph API method.
url="http://localhost:8080/v1/relationships/getRelationshipGraph"

# Execute the getRelationshipGraph API method.

curl -s -X GET --header 'Accept: application/json' "${url}"
```

The result of the call is a JSON representation of a graph:

```
{
  "durationInMicroseconds": "77303",
  "result": {
    "nodes": [
      {
        "NodeID":
"QmXBzmEa6N7Wm4m45sxo3q5RfYNU647x6KhxZVnJYjdjGM",
        "FileID": "randomArtifact1_20220818T142058Z.bin"
      }
    ],
    "edges": [
      {
        "NodeIDA":
"QmXBzmEa6N7Wm4m45sxo3q5RfYNU647x6KhxZVnJYjdjGM",
        "NodeIDB":
"QmXBzmEa6N7Wm4m45sxo3q5RfYNU647x6KhxZVnJYjdjGM"
      }
    ]
  }
}
```

Python scripting available in the source code repository (tools/digraphGenerator.py) converts this JSON representation into a GraphViz-compatible graph file

```
digraph "Create One Artifact" {
 graph [area=3 ranksep=3 ratio=auto]
 QmXBzmEa6N7Wm4m45sxo3q5RfYNU647x6KhxZVnJYjdjGM
[label="randomArtifact1_20220818T142058Z.bin\n(Object ID:
QmXBzmEa6N7Wm4m45sxo3q5RfYNU647x6KhxZVnJYjdjGM" color=black
fillcolor=gray fontcolor=black shape=box style=filled]
 QmXBzmEa6N7Wm4m45sxo3q5RfYNU647x6KhxZVnJYjdjGM ->
QmXBzmEa6N7Wm4m45sxo3q5RfYNU647x6KhxZVnJYjdjGM [weight=0.1]
}
```

When used as input to a GraphViz generator (such as https://dreampuf.github.io/GraphvizOnline) using the “dot” engine, the result is as shown in Fig. 6. The graph shows the artifact’s self-relationship, its specified identifier, and the identifier generated for it by the DFS.

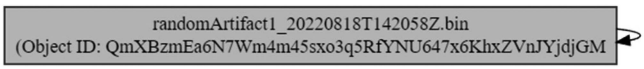

FIGURE 6. Result of graphing the directed graph of a single self-related artifact using the GraphViz "dot" engine.

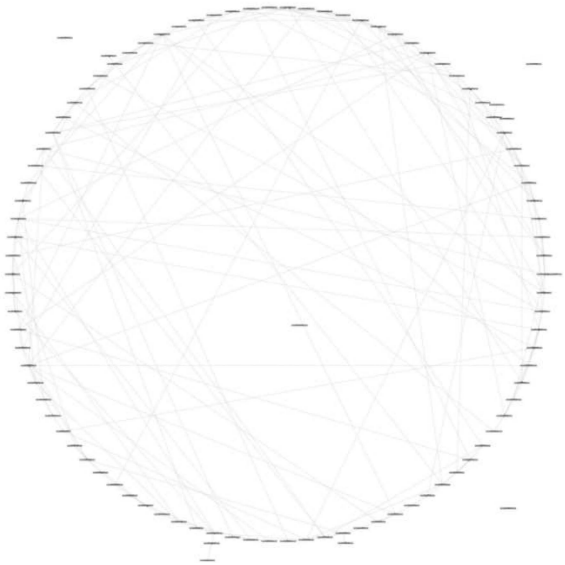

FIGURE 7. Result of graphing the directed graph of 100 randomly related random artifacts using the GraphViz "circo" engine.

FIGURE 8. Memory usage remains relatively flat at less than 4 GiB, whereas CPU usage rises from its steady state of 0.25 cores to 1.3 cores during processing.

To demonstrate a more visually interesting result, a Linux Shell loop construct can be used to generate multiple random artifacts and further randomize their relationships. Fig. 7 shows the result of creating 100 randomly related random artifacts.

VII. PERFORMANCE AND VISIBILITY

An important aspect of evaluating this application is confirming that its resource usage behaves as expected and ensuring searchable, consistent tracing is available to validate the application's behavior. The Kubernetes dashboard provides visibility into the CPU and RAM usage of the application as it executes. Fig. 8 shows the behavior of the system before and after generating numerous artifacts and relationships.

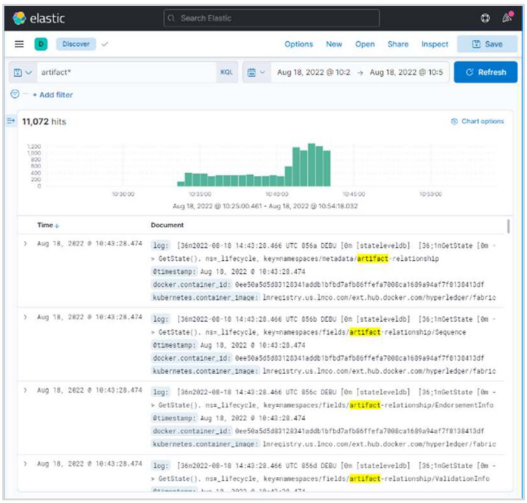

FIGURE 9. Logging from the artifact–relationship smart contract as shown in the elastic dashboard.

As shown in Fig. 9, the Fluentd processing collects information about the execution of each smart contract and Kubernetes-based element of the application.

VIII. CONCLUSION

Applying ChaordicLedger to the process of developing large-scale integrated systems is applicable before, during, and after development, though the greatest benefit is realized when it is employed at the beginning of a development project; building the knowledgebase from the start ensures that the concept of knowledge transfer becomes a part of the development process, not something that is added-on after the fact. By leveraging this application and applying custom business rules through smart contracts, businesses can embed and source-control their business rules for CM and system verification/validation, further removing the risk of human error and subjectivity. The distributed nature of the ledger and the file system allows distributed and colocated teams equitable access to the same knowledge base without requiring centralized management. ChaordicLedger is a contextualized knowledgebase that treats artifacts and their relationships as first-class elements of its API, and as such, the data are accessible to all stakeholders and any process automation tools in use by the teams; this provides every interested party with a consistent, actionable, and objective view of the system's state of verification and validation.

REFERENCES

[1] IBM, "Overview of rational DOORS," 2022. Accessed: Sep. 14, 2022. [Online]. Available: https://www.ibm.com/docs/en/ermd/9.6.0?topic=overview-rational-doors

[2] ALDEC, "Requirement verification traceability matrix," 2022. Accessed: Sep. 14, 2022. [Online]. Available: https://www.aldec.com/en/solutions/requirements_management/traceability--requirement-verification-traceability-matrix
[3] M. F. Marchini, "ChaordicLedger: Digital transformation and business intelligence via data provenance and ubiquity," in *Proc. IEEE Int. Syst. Conf.*, 2022, p. 4.
[4] M. F. Marchini, "Distributed ledgers in developing large-scale integrated systems," in *Proc. IEEE Int. Syst. Conf.*, 2021, p. 7.
[5] Scaled Agile, Inc., "Applying SAFe to hardware development—Scaled agile framework," 2022. Accessed: Aug. 23, 2022. [Online]. Available: https://www.scaledagileframework.com/applying-safe-to-hardware-development/
[6] Agile Alliance, "Agile manifest for software development," 2022. Accessed: Sep. 14, 2022. [Online]. Available: https://www.agilealliance.org/agile101/the-agile-manifesto/
[7] B. E. White, "Leadership under conditions of complexity," in *Proc. 11th Syst. Syst. Eng. Conf.*, 2016, p. 3.
[8] E. Mathisen, K. Ellingsen, and T. Fallmyr, "Using business process modelling to reduce the effects of requirements changes in software projects," in *Proc. 2nd Int. Conf. Adaptive Sci. Technol.*, 2009, pp. 14–19.
[9] E. Dou, "Cost model for verifying requirements," in *Proc. IEEE Int. Autom. Testing Conf.*, 2016, p. 1.
[10] "COCOMO II—Constructive cost model." 2022. [Online]. Available: http://softwarecost.org/tools/COCOMO/
[11] Defense Information Systems Agency (DISA), "DevSecOps—DoD cyber exchange," 2022. Accessed: Aug. 25, 2022. [Online]. Available: https://dl.dod.cyber.mil/wp-content/uploads/devsecops/img/DevSecOps-Software-Lifecycle.png
[12] U.S. Department of Defense, "Chief information officer— U.S. Department of Defense," 2019. Accessed: Aug. 26, 2022. [Online]. Available: https://dodcio.defense.gov/Portals/0/Documents/DoD%20Enterprise%20DevSecOps%20Reference%20Design%20v1.0_Public%20Release.pdf
[13] N. Ramasubbu and R. K. Balan, "Overcoming the challenges in cost estimation for distributed software projects," in *Proc. 34th Int. Conf. Softw. Eng.*, 2012, p.–91.
[14] Hyperledger, "Architecture reference," 2021. Accessed: May 10, 2021. [Online]. Available: https://hyperledger-fabric.readthedocs.io/en/latest/architecture.html
[15] "IPFS is the distributed web," 2021. [Online]. Available: https://ipfs.io
[16] K. Kohli, "IPFS + blockchain = decentralised file storage," Coinmonks, 2018. [Online]. Available: https://medium.com/coinmonks/ipfs-blockchain-decentralised-file-storage-9ef3a1fa307b
[17] Globalcoin, "Decentralized file storage for blockchain: TCFS vs IPFS," 2018. [Online]. Available: https://globalcoinreport.com/decentralized-file-storage-for-blockchaintcfs-vs-ipfs/
[18] National Aeronautics and Space Administration, "Technology readiness level," 2021. Accessed: Aug. 25, 2022. [Online]. Available: https://www.nasa.gov/directorates/heo/scan/engineering/technology/technology_readiness_level
[19] *IEEE Standard for System, Software, and Hardware Verification and Validation*, IEEE Std. 1012-2016, 2016.
[20] B. Dunbar, "Technology readiness level," 2012. [Online]. Available: https://www.nasa.gov/directorates/heo/scan/engineering/technology/technology_readiness_level
[21] OpenAPI Initiative, "OpenAPI specification," 2020. [Online]. Available: http://spec.openapis.org/oas/v3.0.3
[22] Scaled Agile, Inc., "Scaled agile framework—SAFe for lean enterprises," 2022. [Online]. Available: https://www.scaledagileframework.com/
[23] "Manifesto for agile software development," 2022. [Online]. Available: https://www.agilealliance.org/agile101/the-agile-manifesto/

MICHAEL F. MARCHINI received the B.Sc. degree in computer engineering from Penn State Erie, The Behrend College, Erie, PA, USA, in 2007, and the M.Sc. degree in software engineering from the Penn State Great Valley School of Graduate Professional Studies, Malvern, PA, USA, in 2009. He is currently working toward the Ph.D. degree in engineering at Villanova University, Villanova, PA, USA.

He is a Principal Computer Systems Architect with Lockheed Martin Space, King of Prussia, PA, USA. He is currently researching the applicability of distributed ledger technology and distributed file systems to the creation of large-scale, integrated systems. His work has been published in the proceedings of the International Systems Conferences in 2021 and 2022.