RESEARCH-ARTICLE

Quantitative Evaluation of Git-Blockchain Synchronization Models for Trustworthy Software Provenance in Internet of Vehicles

IWINOSA AIDEYAN, Clemson University, Clemson, SC, United States

MERT D. PESÉ, Clemson University, Clemson, SC, United States

RICHARD. R. BROOKS, Clemson University, Clemson, SC, United States

Open Access Support provided by:

Clemson University

Published: 13 March 2026
Accepted: 15 January 2026
Revised: 11 January 2026
Received: 28 August 2025

Quantitative Evaluation of Git-Blockchain Synchronization Models for Trustworthy Software Provenance in Internet of Vehicles

IWINOSA AIDEYAN, Clemson University, USA
MERT D. PESÉ, Clemson University, USA
RICHARD. R. BROOKS, Clemson University, USA

The Internet of Vehicles (IoV) relies on frequent over-the-air (OTA) software updates to deliver new features, patch vulnerabilities, and maintain safety. As vehicles evolve into cyber-physical platforms, protecting software supply chains (SSCs) becomes a critical challenge.Existing blockchain applications in automotive supply chains focus on hardware provenance and component traceability, leaving software delivery pipelines undersecured. Git, the backbone of collaborative development, lacks tamper resistance: features such as rebase and force-push enable history rewriting, while compromised credentials or CI/CD tokens allow adversaries to poison repositories. In IoV settings, a single breach can cascade through multi-tier suppliers, exposing safety-critical functions. Despite blockchain's potential to provide immutability and auditability, no prior study has systematically evaluated synchronization strategies for integrating Git with distributed ledgers.

We define and implement three Git–blockchain synchronization models (independent clone, differential patch, and continuous direct push) on a permissioned blockchain testbed. Using a real automotive codebase, we conducted 270 experimental runs and applied statistical methods including pairwise Z-tests with Holm–Bonferroni correction, effect size analysis, correlation, and regression modeling. Results show that disk footprint is the dominant predictor of synchronization time. Model 1 consistently achieved superior efficiency, while Models 2 and 3 offered viable trade-offs for bandwidth-constrained and compliance-driven environments. All models achieved 100% consistency, proving the feasibility of blockchain-backed provenance in IoV pipelines. This study establishes foundational principles for Git–blockchain integration and introduces GuixChain, a future end-to-end framework unifying Git, blockchain, reproducible builds, SBOMs, and secure OTA deployment to deliver trustworthy, transparent, and continuously verifiable automotive SSCs.

CCS Concepts: • **Security and privacy → Software security engineering**; • **Software and its engineering → Design patterns**; • **Computer systems organization → Embedded software**.

Additional Key Words and Phrases: Software Supply Chain Security, Git-BlockchainIntegration, Hyperledger Fabric, Internet-of-Vehicles, Blockchain, Version control, Git, Software security
DISTRIBUTION STATEMENT A. Approved for public release: distribution is unlimited. OPSEC #10010

1 Introduction

1.1 Overview

The Internet of Vehicles (IoV) is transforming cars into interconnected cyber-physical platforms. Vehicles rely on frequent over-the-air (OTA) updates to add features, patch vulnerabilities, and improve safety. This flexibility expands the attack surface, threatening both passenger safety and intellectual property (IP). A compromised update pipeline allows adversaries to inject malicious code, reverse engineer designs, or disrupt systems.Although

Authors' Contact Information: Iwinosa Aideyan, Clemson University, Clemson, USA, iaideya@clemson.edu; Mert D. Pesé, Clemson University, Clemson, USA, mpese@clemson.edu; Richard. R. Brooks, Clemson University, Clemson, USA, rrb@clemson.edu.

ACM 2769-6480/2026/3-ART
https://doi.org/10.1145/3800581

user privacy is a concern, the foremost risks are protecting code provenance and ensuring only trustworthy software reaches electronic control units (ECUs).

Supply chain attacks have become 'the new normal' in cybersecurity, as noted by NIST's 2024 threat assessment [12]. High-profile incidents illustrate these threats. The SolarWinds Sunburst attack compromised over 18,000 organizations, with remediation costs exceeding $100 billion. Attackers maintained access for 9 months, exfiltrating data and source code [2]. The Log4j vulnerability (CVE-2021-44228) exposed millions of systems to remote code execution [37]. In the automotive contexts, such breaches will let attackers push malicious OTA updates that disable safety systems, manipulate Advanced Driver-Assistance System (ADAS) calibration parameters, or brick entire fleets. Recent incidents underscore this reality: In October 2025, a faulty OTA update bricked thousands of Jeep vehicles, leaving owners stranded [57]. The 2022 Toyota supplier breach exposed source code for T-Connect services, potentially enabling remote vehicle access [44]. Build chain attacks such as the XZ Utils backdoor discovered in 2024 nearly compromised Linux distributions worldwide, including automotive Linux systems [CVE-2024-3094] [10]. Verifiable provenance and tamper-proof audit trails are essential to safeguard vehicle functionality.

Git underpins collaborative development but a compromise from single privileged user can bypass the forge's protections while evading detection [67]. Features such as `rebase` and `force-push` enable history rewriting, allowing attackers to erase or replace evidence of tampering. In IoV, Git orchestrates software development across five phases, illustrated in Figure 1. The phrases are development (OEMs and Tier 1/2 [1] suppliers collaborate on ECU firmware, AUTOSAR [2] components, and infotainment through shared repositories), integration, certification, deployment and maintenance. Figure 1 shows how Git serves as the backbone for collaboration during Development, Integration and Maintenance. Each transition point presents unique vulnerabilities from commit poisoning during development to rollback attacks during maintenance, demonstrating how a single compromised Git credential can cascade malicious code through the entire supply chain.

Attackers gain Git access through multiple vectors, including compromised credentials, stolen SSH keys or tokens, supply chain attacks on continuous integration and continuous delivery(CI/CD) accounts[3], insider threats, and misconfigured repositories. A compromised account can poison feature branches, tamper with Git hooks, or abuse CI/CD tokens with elevated privileges. The 2021 Codecov breach [49] exemplifies this, where attackers modified a CI/CD script to harvest Git credentials from hundreds of companies. In automotive supply chains, a single compromised Tier 2 account can propagate malicious code across all upstream partners. This creates openings for ransomware, rollback attacks, or subtle deviations that infiltrate OTA workflows unnoticed. In compromised CI/CD pipelines, such attacks propagate silently, enabling remote control, data theft, or sabotage. Without blockchain's immutable audit trail, these compromises remain undetectable until catastrophic failure. Our Git-blockchain integration provides the cryptographic verification needed to detect tampering at any point in this exponentially vulnerable chain."

Distributed ledger technologies (DLTs) and blockchains provide immutability, consensus, and tamper-evident logs [47]. Blockchain adoption in automotive supply chains has progressed in traceability, transparency, and trustworthiness [27]. Prior research shows how blockchains enhance transparency in physical supply chains and secure artifacts such as version histories or bills of materials [47]. However, no study has integrated Git workflows with a permissioned blockchain to verify real-time commits in continuous integration environments for connected vehicles. Moreover, existing work has not systematically compared synchronization approaches to quantify their impact on auditability and resource efficiency. By synchronizing Git operations with blockchain transactions,

[1]Tier 1 suppliers deliver complex subsystems (braking, infotainment), and Tier 2 suppliers provide components or modules to Tier 1.

[2]AUTomotive Open System ARchitecture, is a global development partnership creating a standardized software framework for ECUs in vehicles.

[3]CI/CD is a set of practices that automates the process of software development, from code integration to deployment.

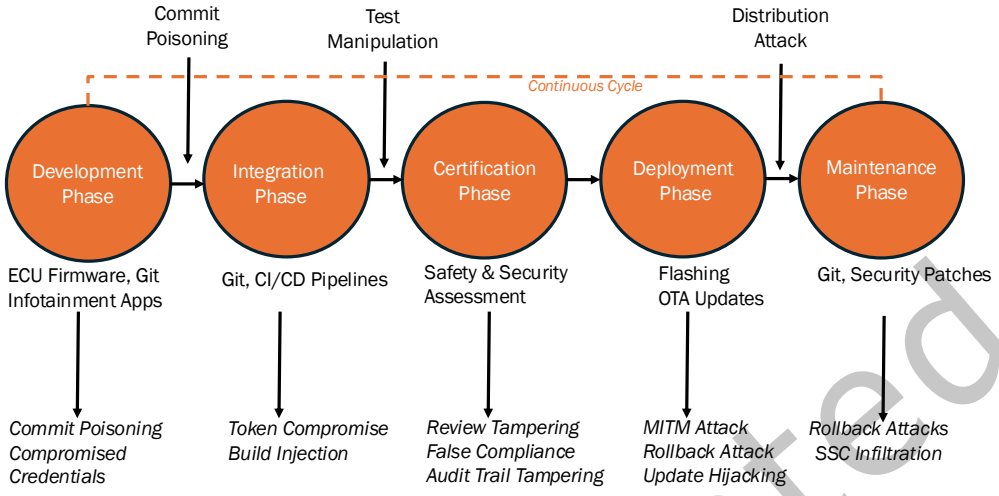


Fig. 1. Automotive Software Lifecycle with Git integration and Attack Vectors

each code modification can be independently validated and permanently recorded, ensuring accountability across OEMs and suppliers. Yet, the performance trade-offs of different synchronization models remain underexplored.

Unlike general DevOps, IoV presents unique constraints that magnify the importance of Git–Blockchain synchronization. OTA updates must meet strict latency requirements for fleet-scale maintenance. Software artifacts directly govern safety-critical functions such as braking, steering, and ADAS, raising the cost of integrity failures far beyond enterprise software. The multi-tier supplier ecosystem means tampering in one software update can cascade across an entire vehicle or a particular OEM model.

This paper addresses these challenges by introducing and evaluating three Git–Blockchain synchronization models. We experimentally compare an independent clone model, a differential patch model, and a continuous direct-push model. Using an automotive codebase on a permissioned blockchain testbed, we conduct ninety controlled runs per model, measuring synchronization time, CPU utilization, memory use, disk consumption, and network throughput. Pairwise Z-tests, Holm–Bonferroni correction, and regression analysis isolate dominant predictors of performance and clarify how synchronization choices affect provenance integrity. The contributions of this work are fourfold:

- Formal Synchronization Models for Multi-Party IoV Supply Chains. We define and implement three Git-Blockchain synchronization architectures tailored to multi-stakeholder environments.
- Empirical Performance Evaluation. We conduct the first controlled experimental comparison of Git-Blockchain synchronization strategies using an automotive codebase deployed on Hyperledger Fabric testbed.
- Statistical analysis: We apply Z-tests, Holm–Bonferroni correction, Cohen's d effect size analysis and regression modeling to isolate the primary predictors of synchronization efficiency.
- Actionable design insights for IoV deployments. We provide evidence-based recommendations for choosing synchronization strategies that address specific constraints while balancing performance efficiency with forensic auditability.

By quantifying the benefits and costs of different Git-Blockchain integration approaches, this study advances the design of trustworthy software provenance systems for connected vehicles.

2 Background

This section reviews the foundational concepts that underpin this study, beginning with the security challenges in IoV software supply chains(SSCs) and the limitations of traditional version control. We then examine how DLTs provide immutability and auditability, before exploring the role of permissioned blockchains.

2.1 Automotive SSC Security

The SSC for the IoV encompasses all processes, from code creation and integration to deployment and maintenance of vehicle ECUs. Modern vehicles depend on OTA updates to introduce new features and patch vulnerabilities. Consequently, each external library, build server, or update channel represents a potential attack vector. O'Donoghue *et al.* show that vulnerabilities in a single component can propagate downstream, leading to widespread compromise of products and services [45].

High-profile incidents such as the SolarWinds Orion breach, which affected over eighteen thousand organizations, including government agencies, demonstrate how a trusted update channel can be subverted to distribute malicious code at scale [2, 36]. Similarly, studies estimate that more than ninety percent of organizations have experienced at least one supply chain incident in the past two years, with projected losses escalating toward $138 billion by 2031 [46].

The intertwined links between OEM and their supplier chain naturally complicate the automotive SSC. OEMs frequently incorporate codebases, templates, or software modules from Tier 1 and Tier 2 suppliers, customizing them for branding and performance requirements. This collaborative development strategy creates challenges in configuration management, change control, and traceability. Many procedures remain inconsistently documented or reported [16, 54], leaving gaps that adversaries can exploit. The Log4j vulnerability (Log4Shell) demonstrated how flaws in widely used libraries could enable remote code execution across diverse applications and systems [20]. In the automotive sector, a vulnerability introduced by one supplier could affect multiple ECUs, undermining safety-critical functions and threatening vehicle reliability. Securing the SSC is therefore essential to ensure provenance and trust, underscoring the need for end-to-end integrity and auditability of artifacts.

2.2 Provenance and Auditability with Blockchain

Blockchain provides immutable records that support verifiable timestamps and digital signatures [47]. At the data layer, blocks link to previous blocks via cryptographic hashes, forming an append-only chain whose integrity is evident when any prior block is altered [41]. Blockchain technology is categorized into three types: public, private, and permissioned blockchains [8]. Each type has distinct characteristics and applications, making them suitable for different use cases. Public blockchains are decentralized networks that allow anyone to participate without permission. They are often associated with cryptocurrencies like Bitcoin and Ethereum, which utilize a permissionless approach to enable open access and transparency [52]. Private or consortium blockchains are restricted networks with limited access to specific participants. Organizations typically use these blockchains, which require more control over the network and its participants. They offer enhanced privacy and control, making them suitable for enterprise applications where data confidentiality is crucial [38]. Permissioned blockchains are a hybrid model that combines elements of both public and private blockchains. They require permission to join the network and participate in the consensus process, but they can be public or private. This type of blockchain is used in enterprise settings where a balance between transparency and privacy is needed [48]. They are deployed between verified organizations and are designed to protect sensitive information while allowing efficient transaction processing [59]. Leading permissioned blockchain platforms include Hyperledger Fabric,

Corda, and Quorum, which differ in consensus mechanisms, modularity, language support, privacy features, and transaction rates [40].

The consensus layer ensures that a majority of validator nodes agree on each block, preventing a single malicious node from corrupting the ledger state [11]. Smart contracts at the contract layer execute predefined logic when conditions are met, enabling automated validation of transactions [56]. A smart contract is a self-executing agreement in which the terms between the parties are directly written into code. This executable code is deployed on a blockchain and stored on a distributed decentralized ledger. Once deployed, a smart contract operates on predetermined conditions. When these conditions are met, the contract automatically executes the corresponding actions, such as transferring digital assets, recording data, or triggering subsequent smart contracts without intermediaries [6]. This automation reduces potential human errors and delays and enhances trust among parties by providing transparency and immutability. Together, these layers create a tamper-evident provenance mechanism.

2.3 Version Control Pipeline

Early version control systems, such as Source Code Control System (SCCS) and Revision Control System (RCS), introduced centralized models where developers checked out code from a central repository and committed changes back [15]. While this enabled collaboration, it suffered from single points of failure and server dependency. Distributed version control systems (DVCS) such as Git overcame these issues by allowing each developer to maintain a full repository copy, enabling offline work, rapid branching, and efficient merging [18, 63]. Developed by Linus Torvalds in 2005, Git has since become the de facto standard in modern software engineering [15]. Its success derives from scalability to large codebases and its rich feature set for collaborative workflows [63].

Git also supports workflow automation via *hooks*. Hooks are user-defined scripts triggered by repository events [6], classified as client-side (executing locally) or server-side (executing on a Git server) [21, 58]. Client-side hooks include `pre-commit`, which can enforce code style or run unit tests before finalizing a commit, and `pre-push`, which validates changes before pushing them upstream. Server-side hooks include `pre-receive`, which can enforce branch-specific access controls or scan commits for vulnerabilities, and `post-receive`, which can trigger automated deployments or notifications. Table 1 lists common client-side hooks, which are the focus of this work.

Table 1. Client-Side Git Hooks and Their Descriptions

Client-Side Hook	**Description**
Pre-commit Hook	Performs code style checks or runs unit tests before a commit is finalized.
Pre-push Hook	Validates changes before they are pushed to a remote branch.
Post-commit Hook	Executes after a commit is created, often for notifications or non-blocking tasks.
Post-push Hook	Executes after a successful push, typically to trigger downstream actions such as builds or alerts.

Git hooks can enforce policies at commit or push time, but they operate in trusted environments and can be bypassed. For example, Developers can skip hooks using git commit –no-verify, direct repository access allows editing .git/hooks/ to disable security checks, force pushes (git push –force) can overwrite protected branches if misconfigured and CI/CD service accounts often bypass hook restrictions entirely. The Codecov

breach demonstrated this vulnerability when attackers modified CI scripts to disable security hooks and exfiltrate credentials [49]. Torres-Arias *et al.* describe "teleport" and rollback attacks that exploit Git references to hide malicious changes [62], while Wheeler *et al.* observe that attackers with sufficient access can rewrite history to erase evidence of code injection [65]. As a result, version control systems alone cannot guarantee provenance or auditability.

3 Related Work

This section reviews key studies on blockchain integration in supply chains and version control systems, highlighting their advances and identifying gaps that this paper addresses. The review is organized into four parts: blockchain in general supply chain management, blockchain integration with version control, blockchain use in CI/CD for software provenance and blockchain applications in the automotive domain.

3.1 Blockchain in Supply Chain Management

Ahmad *et al.* [3] classify blockchain adoption drivers into product traceability, process automation and customer experience, noting benefits such as improved transparency, security and efficiency across logistics, pharmaceuticals and manufacturing. Tokkozhina *et al.* [60] use a systematic review to demonstrate that smart contracts reduce human error, lower transaction costs, enable real-time auditing, and acknowledge scalability and privacy barriers. Kouhizadeh and Sarkis [33] apply Porter's value chain framework to show blockchain's potential for sustainable supply chains but highlight challenges with regulatory compliance and energy consumption. These works establish blockchain's versatility in physical supply chains but do not extend to software artifacts or developer workflows.

3.2 Blockchain and Version Control

Previous work has explored various synchronization strategies for distributed version control. Git's native synchronization includes git pull, git fetch, and git clone . Enterprise Git platforms employ hooks and webhooks for event-driven synchronization [GitHub Actions [17], GitLab CI [14]]. Several prototypes integrate blockchain with version control systems. Torres-Arias *et al.* [62] identify metadata manipulation attacks in Git and propose a signed reference state list to detect tampering. Nizamuddin *et al.* [43] combine Ethereum smart contracts with InterPlanetary File System (IPFS) to track document versions, storing hashes on-chain for integrity. Grilli and Speziali [25] use periodic Merkle tree snapshots for repository state verification but don't address real-time synchronization. Hammad *et al.* [26] propose off-chain storage with on-chain anchors but lack comparative evaluation of synchronization strategies. Our contribution is the first systematic comparison of three distinct synchronization patterns specifically optimized for blockchain integration, with quantitative evaluation of their trade-offs in automotive contexts [13].

3.3 Blockchain for Software Supply Chain Security and CI/CD

Recent research has explored blockchain integration into software development pipelines to enhance build provenance, deployment security, and audit traceability. Saleh et al. [53] propose a blockchain-based framework for securing continuous integration and deployment pipelines in cloud environments, using a "separation of concern" approach that modularizes CI/CD entities as blockchain blocks to contain threats locally. Their work demonstrates blockchain's potential for preventing unauthorized pipeline modifications but focuses on generic cloud CI/CD rather than automotive-specific constraints such as safety-critical OTA updates or multi-party supplier distrust. Similarly, Kalyan [30] presents a DevSecOps pipeline with decentralized, tamper-evident audit trails using smart contracts, Jenkins, Docker, and Truffle for real-time verifiable logging. While these approaches

validate blockchain for CI/CD provenance, they do not address Git synchronization performance trade-offs or compare alternative integration architectures which is the focus of our work.

The in-toto framework [61] provides supply chain integrity through cryptographically signed metadata describing each pipeline step (commits, builds, tests, deployments), enabling verification that software artifacts result from expected processes. In-toto's layout files define trust relationships and required functionaries, creating an auditable chain-of-custody. However, in-toto relies on centralized layout management and does not employ distributed ledger consensus, making it vulnerable to single points of compromise. Google's SLSA (Supply chain Levels for Software Artifacts) framework [55] defines maturity levels for supply chain security, prescribing controls such as version-controlled builds, hermetic compilation, and non-falsifiable provenance. While SLSA provides a conceptual roadmap, it does not mandate specific implementation technologies. Grafeas [23] and Sigstore [35] offer centralized artifact metadata storage and transparency logs, respectively, but lack the Byzantine fault tolerance and decentralized verification that permissioned blockchains provide for multi-party ecosystems.

3.4 Blockchain in the Automotive Industry

Blockchain has also been applied in automotive supply chains for parts provenance and maintenance records. Reddy et al. [50] review permissioned platforms for traceability in volatile environments, while Kamble et al. [31] empirically show blockchain's positive impact on supply chain integration in Indian automotive firms. Practical systems such as VinChain [64] and ClassicsChain [39] extend these ideas.

VinChain creates an immutable vehicle history record, including manufacturing details, ownership transfers, maintenance logs, accident reports, and parts replacements [64]. Smart contracts automate interactions and enforce compliance, ensuring reliable, real-time updates and auditability. Manufacturers use VinChain to verify parts' authenticity, dealers gain trustworthy vehicle histories, and consumers benefit from enhanced transparency. ClassicsChain, tailored to classic car restorations, uses Hyperledger Fabric to securely and immutably record technical specifications, restoration procedures, certifications, ownership histories, and other essential documentation [39].

Despite their utility, these solutions focus on physical components and ownership records. They do not extend to software audit trails, underscoring a gap that our research addresses.

3.5 Research Gap

Automotive blockchain solutions have concentrated primarily on hardware parts provenance, maintenance logs, and ownership records, leaving software delivery largely unexplored. Prior work in software supply chains has secured bills of materials or document snapshots on-chain [26, 43], and prototype frameworks have wrapped Git repositories into storage units for periodic anchoring [25]. However, these approaches treat the ledger as an external audit log rather than a native extension of developer workflows.

In parallel, blockchain-enabled CI/CD research has advanced with solutions like in-toto, SLSA, and Grafeas providing provenance-aware security models. Yet these studies assume trusted organizational boundaries and do not account for the adversarial dynamics of automotive supply chains, where multiple OEMs and Tier 1/Tier 2 suppliers must coordinate software under conditions of partial trust.

To our knowledge, no prior work has integrated Git itself with a permissioned blockchain to record and verify each commit in real time, nor has any study empirically compared alternative synchronization architectures to quantify their impact on CPU, memory, disk, network, and latency metrics. Our work fills this gap by proposing three distinct Git–Blockchain synchronization models and conducting the first experimental comparison of their resource overhead, auditability, and scalability trade-offs specifically for IoV software supply chains.

4 Synchronization Models

This section presents the three Git-Blockchain synchronization models evaluated in this study. Each model defines a distinct workflow for recording commit metadata on a permissioned ledger. We describe the architecture, workflow steps, and integration points for each model.

4.1 Scenario 1: Independent Clone

In this model, each node in the network maintains an up-to-date copy of the repository by performing a full clone. Developers commit and push changes using standard Git commands. A blockchain client application is invoked on both commit and push events to extract metadata such as commit hash, author, timestamp, message and repository identifier, and submit this data as transactions to a Hyperledger Fabric network as illustrated in Figure 2. The smart contract implements two functions: `CreateGitCommit`(commitHash, author, timestamp, message) which stores each commit in the ledger, and `HandleGitPush(commitHash, branch, remoteUrl)`, which verifies the commit and emits a `SyncRepo` event when a push is recorded. All nodes subscribe to this event and, upon receiving it, remove their local clone and execute a fresh `git clone`, ensuring that every participant holds an identical repository state. After cloning, each node verifies that the latest commit matches the ledger record, triggering an alert if discrepancies arise. Note that all smart contract functions are explained further in section 5.

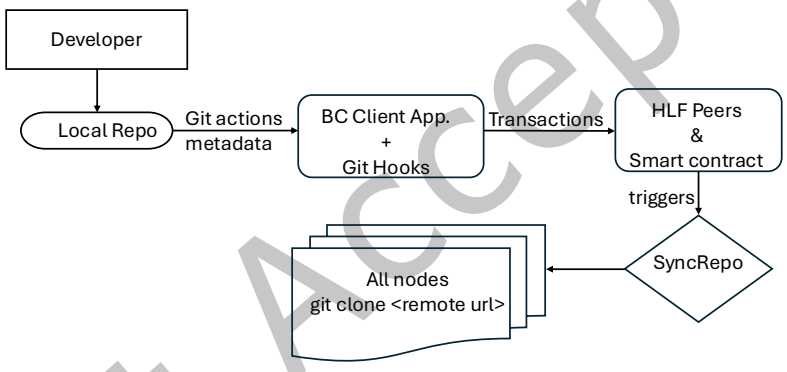


Fig. 2. Model 1: Independent clone with Git actions triggering a full clone on all nodes.

Workflow.

(1) Developer runs `git commit`; post-commit hook invokes the blockchain client, which submits a `CreateGitCommit` transaction.
(2) Developer runs `git push`; post-push hook invokes the client, which submits a `HandleGitPush` transaction.
(3) The smart contract orders and commits the push transaction, then emits `SyncRepo`.
(4) All nodes detect `SyncRepo`, delete their local clone and perform `git clone <repo-url>`.
(5) Each node validates that the newly cloned repository's latest commit hash matches the ledger entry.

Scenario 1 provides strong auditability, as every commit and push is immutably recorded. The resynchronization trigger ensures consistent repository state across all nodes. However, this approach incurs considerable network and disk overhead due to full clones on each push and may introduce latency for large repositories, potentially disrupting development workflows.

4.2 Scenario 2: Differential patch Repository Synchronization

In this model, each node retains a persistent local clone of the repository. Upon each commit, a blockchain client application captures the commit metadata: hash, author, timestamp, message and repository identifier, and submits a `CreateGitCommit` transaction to the ledger. When the developer executes `git push`, the client submits a `HandleGitPush` transaction, which triggers a smart contract event. The contract then retrieves the difference(diff or delta) between the newly pushed commit and the last recorded commit, packages this delta as a patch file, and emits a `SyncDiff` event containing the diff identifier. All nodes subscribed to this event retrieve the patch from a shared storage location or directly from the transaction payload, apply the patch to their local clone using `git apply` as illustrated in Figure 3, and thus converge to the new repository state without performing a full clone. This approach minimizes data transfer by recording only incremental changes on the blockchain while preserving an immutable audit log of both commits and pushes.

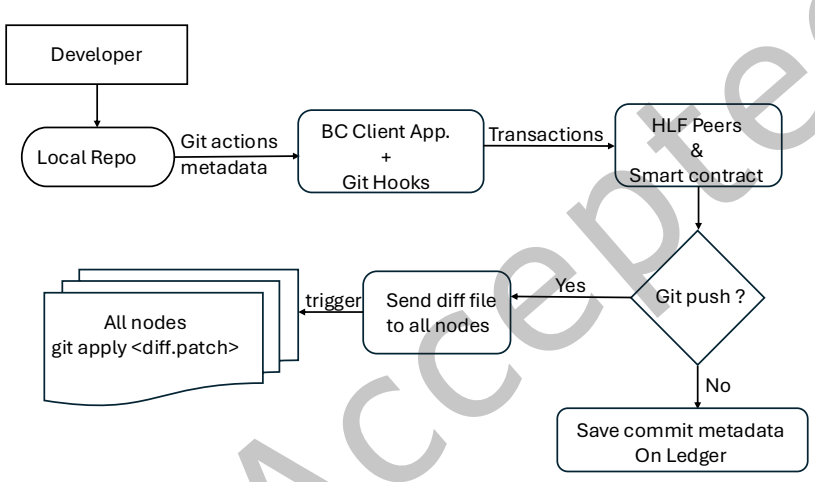


Fig. 3. Model 2: Diff-based synchronization where patches are distributed and applied on all nodes.

Workflow.

(1) Developer runs `git commit`; post-commit hook invokes the blockchain client, which submits a `CreateGitCommit` transaction with commit metadata.
(2) Developer runs `git push`; post-push hook invokes the client, which submits a `HandleGitPush` transaction.
(3) The smart contract processes the push transaction, computes the diff between the newest commit and the last recorded commit, and stores the patch in the ledger or in an off-chain store referenced by the transaction.
(4) The contract emits `SyncDiff` containing the patch reference.
(5) All nodes detect `SyncDiff`, retrieve the patch, apply it to their local clone with `git apply` and confirm that their repository's head matches the commit hash recorded on-chain.

Scenario 2 achieves synchronization with significantly lower network and storage overhead compared to full cloning. By transmitting only the differences, this model supports finer-grained auditability and reduces latency in environments with limited bandwidth. However, maintaining and applying patches adds complexity to the client logic, and large diffs in monolithic repositories may still incur nontrivial data transfer.

4.3 Scenario 3: Continuous Direct Push Synchronization

In this model, every Git commit is streamed to the blockchain in real-time. A client application intercepts each `git commit` event and immediately submits a `CreateGitCommit` transaction containing the commit hash, author, timestamp, message, branch and repository identifier. As soon as the transaction is committed, the smart contract emits a `CommitStreamed` event.All nodes subscribed to this event receive the metadata and automatically perform a fast pull of the updated branch using `git pull`. This ensures that every participant's local clone reflects the latest commit without any additional diff computation or manual synchronization steps. Because commits are sent one by one, the ledger contains a complete, ordered history of development activity as it happens, as shown in Figure 4.

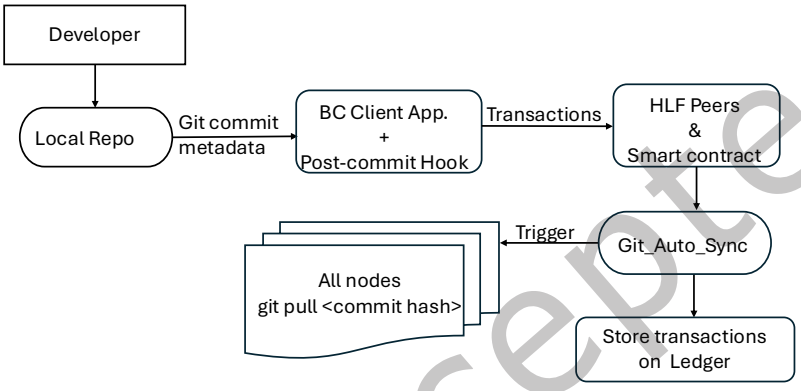


Fig. 4. Model 3: Continuous direct push where each commit is streamed and pulled by all nodes in real time.

Workflow.

(1) Developer runs `git commit`; post-commit hook invokes the blockchain client, which submits a `StreamCommit` transaction to the ledger with full commit metadata.
(2) The smart contract orders and commits the transaction, then emits a `CommitStreamed` event.
(3) All nodes detect `CommitStreamed` and execute `git pull` to fetch the new commit and update their local clone.
(4) Each node verifies that the pulled commit hash matches the metadata recorded in the blockchain.

Scenario 3 maximizes automation and immediacy by streaming each commit as it occurs. It eliminates the overhead of clone or patch operations and provides the most fine-grained audit trail. On the other hand, high commit frequency can increase transaction volume on the ledger and may require efficient batching or rate-limiting strategies to manage performance.
With these three models defined, the next section details the methodology for implementing, instrumenting, and experimentally evaluating each approach under realistic IoV software pipeline conditions.

5 Methodology

This chapter describes the experimental approach to implementing and evaluating the three Git-Blockchain synchronization models.

5.1 Research Questions and Hypotheses

This study evaluates the performance trade-offs of three Git–Blockchain synchronization models in IoV environments. The central research question is: How do different Git-blockchain synchronization strategies impact system performance and resource consumption? We hypothesize that synchronization strategy significantly affects latency (H1), resource consumption across CPU, memory, disk, and network usage (H2), and that resource metrics significantly influence synchronization time (H3). The independent variable is the synchronization model. The dependent variables are synchronization time (seconds) and system resources (CPU utilization, memory usage, disk consumption, and network throughput).

Control variables were held constant to ensure fair comparison. These include repository size (5 MB), commit frequency, hardware configuration (2 GB RAM, 50 GB disk per VM), network topology (five-node permissioned blockchain), and consensus mechanism (SmartBFT). All experiments were executed on a VirtualBox testbed running Ubuntu with Hyperledger Fabric. Each model was executed across 90 independent runs to ensure statistical validity.

5.2 Experimental Design

This study employs a design-based experimental methodology informed by design science principles and iterative refinement. We follow the design–implement–evaluate cycle articulated by Reeves [51] and Hevner *et al.* [28], beginning with a conceptual model, proceeding to a working prototype built atop Git and Hyperledger Fabric, and concluding with evaluation. We aim to evaluate three Git-Blockchain synchronization models for securing software provenance. The models represent alternative ways of synchronizing automotive codebases with a blockchain-backed ledger. We developed a prototype system, executed controlled experiments, and applied statistical analysis to determine performance trade-offs. The research design follows three phases: (1) defining system architecture and components, (2) executing experiments with an automotive software project, and (3) applying statistical evaluation.

First, we define a multi-layer architecture that integrates Git, a client application for metadata extraction and blockchain interaction, and a permissioned ledger that immutably records commit events via smart contracts. Next, we implement the prototype by developing a Go-based client triggered by Git hooks, deploying chaincode on a Fabric network orchestrated with Docker Swarm across five peer organizations, and integrating Jenkins for continuous deployment and automated verification of commit hashes. We then establish data collection pipelines using Prometheus [19] to scrape metrics such as CPU utilization, memory consumption, synchronization latency and network throughput every 30 seconds, visualized in Grafana [34] dashboards. Finally, we evaluate the system quantitatively to assess its auditability, consistency and overhead.

The core of the system is a client application that intercepts each Git action, extracts commit metadata (hash, author, timestamp, commit message), and invokes the Fabric gateway to submit a corresponding transaction. By design, this framework ensures that every commit is cryptographically verifiable and permanently logged, eliminating risks of history rewriting or metadata forgery. Ensuring that all nodes converge to a single, authenticated repository state and that the audit trail supports forensic analysis, regulatory compliance and trust in OTA vehicle updates.

5.3 Statistical Analysis

We applied pairwise two-tailed Z-tests with a significance threshold of $\alpha = 0.05$ to evaluate differences between models. Given 15 comparisons (3 model pairs × 5 metrics), the Holm–Bonferroni correction was applied to control family-wise error rate. Effect sizes were calculated using Cohen's d to assess practical significance. Multiple linear regression examined the relationship between resource predictors and synchronization time, with assumptions verified through residual analysis.

6 System Design and Implementation

This section presents the architecture of the proposed Git–Blockchain framework. It introduces the overall design principles, outlines the layered structure, and details the system components. The subsections describe the repository setup, client application, blockchain network, smart contracts, and supporting infrastructure, followed by the deployment scope of the framework.

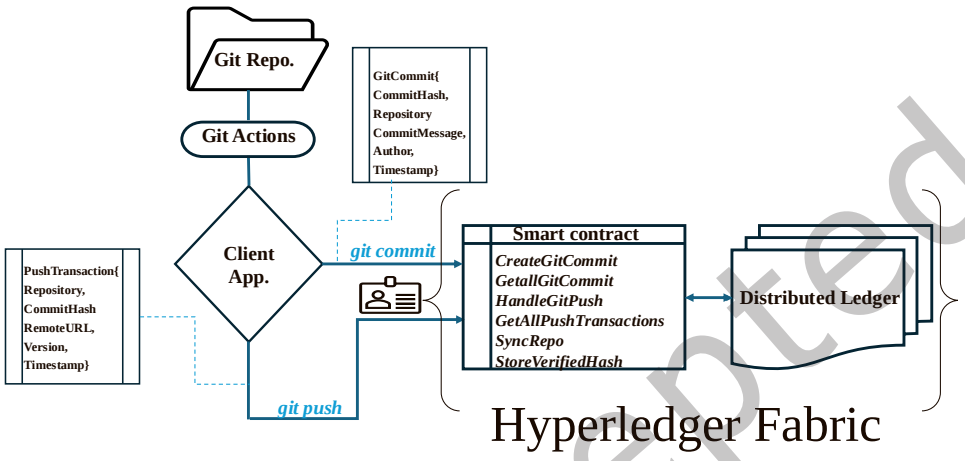


Fig. 5. Multi-layered architecture integrating Git, client application and Hyperledger Fabric to secure software supply chains.

6.1 System Architecture

The proposed framework comprises multiple integrated layers to guarantee end-to-end auditability, consistency and traceability of software changes. Figure 5 shows the architecture with four core components: Git repository and hooks, client application, blockchain network, and smart contract layer. At the bottom, the *Git Repository Layer* serves as the definitive source of truth. Developers perform standard Git operations (`commit`, `push`) against local clones, which preserve full version history and metadata. Sitting above, the *Client Application Layer* intercepts these events via Git hooks, extracts commit identifiers, author information, timestamps, and commit messages, and formats them into transaction payloads. This Go application then invokes the Fabric Gateway API to submit transactions to a permissioned ledger. Finally, the *Blockchain Layer*, realized as a Hyperledger Fabric network, provides an immutable, tamper-proof record of all Git events. Smart contracts validate incoming transactions, enforce endorsement policies, and emit synchronization directives as each synchronization model dictates. These layers ensure that every code change is cryptographically anchored and all nodes converge on a single, verifiable repository state.

6.2 System Components

The following core components instantiate the architecture:

Git Repository. A central Git repository was the entry point for software changes. It was sourced from the NXP S32 DESIGN Studio and is a practical example of an automotive software project. It is an integrated development environment (IDE) sample project for developing embedded applications on the S32K144 microcontroller[4]. The

[4]A device commonly employed in automotive embedded systems.

composition comprises 42 C files into 12 folders. The code demonstrates basic microcontroller operations such as configuring clock signals for peripheral modules, setting up GPIO pins for input and output, reading an input state and controlling an output based on that state to be deployed on an S32K144 evaluation board.

This repository reflects automotive embedded software, providing a dataset for testing blockchain integration, and is stored on all peer nodes. It forms the system's backbone, providing the raw data that is later processed by the client application and immutably recorded on the blockchain network. In section 2.3, we discussed Git hooks, specifically highlighting the two main scripts used in this work: `post-commit` and `post-push`.

Client Application. The custom-developed client application, named gittransfer.go and written in Golang, bridges the Git repository and the blockchain network. The intermediary monitors commit events, plus post-commit and post-push hooks, extracts metadata and triggers blockchain transactions to record each commit. The application calls the blockchain ledger by invoking transactions implemented in the smart contract as illustrated in Figure 6. Upon detecting a Git event, it extracts metadata. It submits either a `RecordCommit`, `RecordPush` or `SyncRepo` transaction depending on the chosen synchronization model to Fabric via the Gateway API to trigger local repository updates.

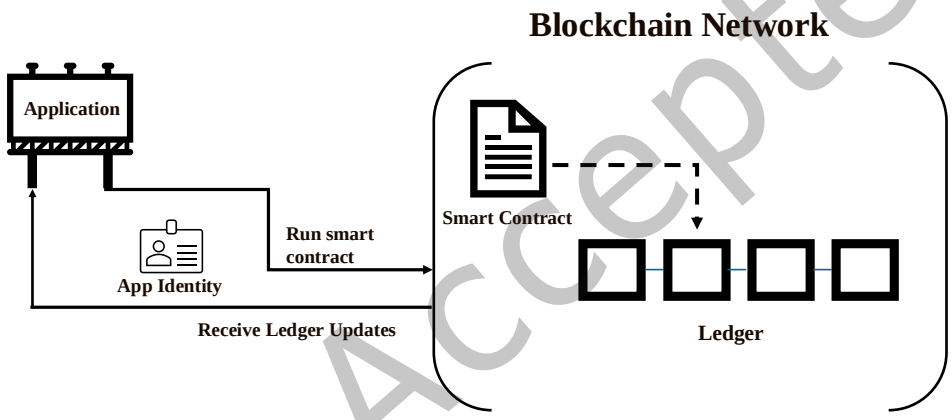


Fig. 6. Client Application interaction with the Blockchain

Additionally, the application demonstrates how to create, update, and query assets on the blockchain without direct access to the smartcontract. Each user must have an application developer identity (X.509 certificates) created by the certificate authorities; otherwise, network access is denied. These identities (tied to each organization) ensure only authorized users/entities can transact with the blockchain. The client application leverages the Fabric Gateway client API. The Gateway helps evaluate a transaction proposal, endorse a transaction proposal, and submit a signed transaction envelope to the ordering service for ledger commitment.

Hyperledger Fabric Network. We employ Hyperledger Fabric (HLF) as our permissioned blockchain platform. Its execute-order-validate architecture separates transaction simulation from ordering and validation, enhancing throughput and reducing nondeterministic behavior [29]. Fabric's modular design supports configurable consensus protocols and private channels, enabling selective data sharing among known participants (OEMs, suppliers, integrators) [66]. Chaincode[5] written in general-purpose languages enforces endorsement policies and embeds business logic directly on the ledger.Compared to public blockchains like Ethereum, Fabric delivers superior

[5]Used interchangeably with smart contract.

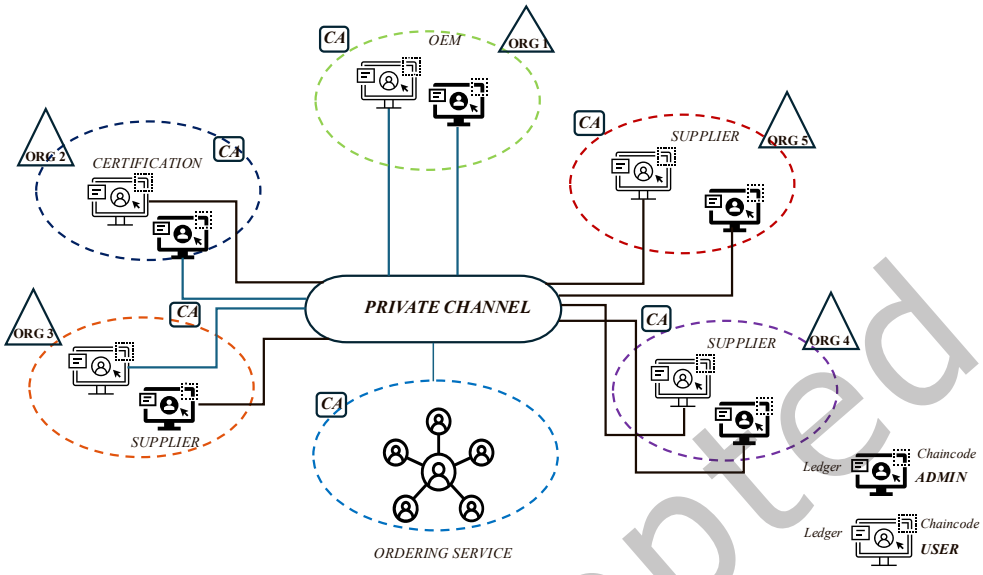


Fig. 7. Hyperledger Fabric Test Network

scalability, privacy control, and performance without high transaction fees or proof-of-work delays [1, 40]. Its permissioned architecture ensures only authorized nodes validate or commit transactions, creating an immutable audit trail without exposing sensitive data to all participants.

Our HLF test network simulates a distributed environment where each node represents different stakeholders (developers, OEMs, suppliers) with private channels for secure communication (Figure 7). An ordering service ensures commit records are correctly sequenced into blocks and committed to the ledger. From transaction ordering through chaincode execution, operations are partitioned to limit required trust and verification across nodes [42].

We employ SmartBFT consensus rather than crash fault-tolerant protocols (Raft, Kafka) because automotive supply chains involve competing stakeholders who may not fully trust one another. Byzantine fault tolerance (BFT) ensures that the system remains secure even if some nodes behave maliciously. BFT also means that attackers need to simultaneously infect more than 1/3 of the nodes to change history. While crash fault tolerant protocols like Raft offer lower latency, they assume all failures are benign (crashes or network partitions), making them unsuitable for multi-party environments where adversarial behavior is possible. SmartBFT's three-phase commit protocol (pre-prepare, prepare, commit) guarantees consistent transaction ordering across honest nodes even when up to $(n-1)/3$ nodes (where n is the number of ordering nodes) act maliciously. While crash fault-tolerant protocols offer lower latency, they assume all failures are benign, making them unsuitable for adversarial contexts. This trade-off prioritizes provenance integrity over raw performance, aligning with automotive safety requirements.

In our SSC implementation, Fabric records each Git operation as a timestamped transaction. The permissioned model ensures only vetted identities can submit or endorse commit metadata, reducing unauthorized modification risks. Automated chaincode invocation validates commit signatures and records metadata immutably. Empirical

studies in food logistics, aircraft maintenance, pharmaceuticals, and automotive component supply chains demonstrate Fabric's ability to handle high transaction volumes with low latency [4, 7, 9], validating its suitability for real-time Git provenance tracking.

Table 2. Smart contract functions, classification, and brief description

Function	**Classification**	**Description**
InitLedger	Initialization	Populates ledger with sample `GitCommit` entries and initializes repository version counters.
SyncRepo	Initialization	Triggers a full repository synchronization (full clone) across peers to bring nodes to the latest committed state.
CreateGitCommit	Storing data	Records a new Git commit on-chain with timestamp and metadata.
HandleGitPush	Storing data	Validates pushed commits, increments repository version, and logs a `PushTransaction`.
StoreVerifiedHash	Storing data	Persists a verified commit hash, verification status, and timestamp (used for later audit).
RecordDiff	Storing data	Stores lightweight diff information under a composite key for delta-synchronization.
SyncDiff	Storing data	Triggers distribution and application of a specific patch/diff across peers to reduce transfer overhead.
ReadGitCommit	Reading data	Retrieves commit metadata (author, message, timestamp) for a given commit hash.
GitCommitExists	Reading data	Checks whether a commit key exists in the world state (used to prevent duplicates).
GetSyncStatus	Reading data	Fetches the latest verification/verification-status record for a repository.
GetAllGitCommits	Process coordination	Returns the full set of recorded commits (useful for audits and historical analysis).
GetAllPushTransactions	Process coordination	Enumerates all recorded push transactions for forensic review and metrics.
GetAllDiffRecords	Process coordination	Lists all stored diff records on the ledger.
GetAllVerifiedCommits	Process coordination	Returns all stored verified-commit entries for chain-wide verification reports.

Smart Contract. It holds facts about a set of assets' current and historical state, forming the heart of the blockchain system. It is an integral part of this work as it governs the recording and verification of commit metadata. Smart contracts verify commit data and write only authenticated records. Once recorded, commits cannot be altered, ensuring a permanent audit trail. Data writing and retrieval are restricted to authorized entities to ensure the integrity and confidentiality of the stored data. The smart contract was written in Golang, testing

its functions and verifying its performance. The smart contract's code base is modular, allowing for further enhancements such as incremental change tracking and real-time verification. We developed the smart contract's function according to initialization, data storage, reading data, and global verification as seen in Table 2.

Auxiliary Infrastructure. A Jenkins server is integrated into the network to support CI/CD automation. The server runs a dedicated Jenkinsfile that orchestrates verification workflows. When a Git push event occurs, Jenkins queries all blockchain nodes to confirm they have synchronized to the same commit hash, then records the verification status on the ledger. Jenkins does not store the repository codebase or blockchain data, it serves purely as an automation orchestrator. The source code resides in Git repositories replicated across all peer nodes, and commit metadata is immutably recorded on the distributed Hyperledger Fabric ledger. Even if Jenkins is compromised or unavailable, the blockchain ledger remains intact and independently queryable by all stakeholders, ensuring that Jenkins does not introduce a single point of failure for provenance integrity.

Prometheus is an open-source systems monitoring and alerting toolkit that collects and stores metrics as time-series data [19], while Grafana is an open-source data visualization and monitoring suite that allows users to create dashboards for visualizing Prometheus metrics [34]. Prometheus scrapes system metrics every 30 seconds, while Grafana renders real-time dashboards for monitoring and post-run analysis.

6.3 Deployment Scope

To clarify the operational scope of this work, the Git-Blockchain synchronization models evaluated in this study operate at the OEM and supplier development infrastructure level, not within vehicle Electronic Control Units (ECUs). The typical deployment architecture comprises two layers:

Development Infrastructure (Synchronization Layer): This layer handles source code provenance, audit trails, and multi-stakeholder collaboration. OEM and supplier development teams maintain Git repositories on enterprise servers or cloud platforms. Git hooks trigger blockchain transactions on a permissioned Hyperledger Fabric network spanning multiple organizational nodes. Continuous integration servers (Jenkins) orchestrate builds, tests, and blockchain verification.

Vehicle Deployment Layer (Out of Scope): After code passes blockchain-verified CI/CD pipelines, final compiled binaries (.hex, .elf) are cryptographically signed. Signed binaries are deployed to vehicle ECUs via OTA update frameworks or diagnostic flashing tools. ECUs do not participate in Git operations or blockchain synchronization. They receive only the final verified software artifacts.

Performance Considerations: The synchronization latency, disk usage, and resource consumption measured in this study reflect the overhead imposed on development workflows, not on vehicle runtime systems. Our experimental testbed emulates a distributed development environment where multiple stakeholders maintain synchronized repositories during collaborative software engineering. The choice of synchronization model affects developer productivity, CI/CD pipeline throughput, and audit granularity, but does not impact vehicle computational resources or safety-critical real-time performance.

7 Experimental Evaluation and Results

This section presents a systematic evaluation of the three Git-blockchain synchronization models. First, we summarize descriptive statistics for synchronization latency and resource consumption in subsubsection 7.3.1. Next, in subsubsection 7.3.2, we explore interdependencies between metrics via correlation maps. We then assess the statistical significance of pairwise differences using two-tailed Z-tests in subsubsection 7.3.3. Finally, we employ multiple linear regression to identify which system resources strongly predict sync time in subsubsection 7.3.5. Together, these analyses offer a comprehensive picture of how each model performs regarding efficiency, resource demands, and scalability

7.1 Experimental Setup

Our testbed comprises five virtual machines (VMs) on a Docker Swarm cluster, each representing a stakeholder in the SSC (OEM, certification authority, two suppliers, integrator). The host is an Intel Core i7, 16 GB RAM, 1 TB SSD, VirtualBox 6.1, Windows 10. Nodes run Ubuntu 20.04 LTS with Hyperledger Fabric v2.5 Docker images. Each node hosts a Fabric peer and a local Git clone. A dedicated ordering service runs SmartBFT on a single channel with a majority endorsement policy. We evaluate each model under identical conditions by replaying sequential commits from the NXP S32K144 project at 30-second intervals. Each model is executed in isolation and repeated 90 runs per model to capture variability.

7.1.1 Model-Specific Execution.

Model 1: Independent clone Synchronization- In this model, every commit and push is recorded on-chain, and a global clone is triggered on all peers. Upon `git commit`, the post-commit hook invokes the smart contract using the line seen in Listing 1.

```
gitTransfer -create -hash "$COMMIT_HASH" -repo "$REPO_NAME" -message "$COMMIT_MESSAGE" -author "
    ↪ $AUTHOR"
```

Listing 1. Post-commit Hook Snippet

This call submits a *CreateGitCommit* transaction. When the developer runs git push, the post-push hook similarly executes Listing 2, which triggers the *HandleGitPush* function. The smart contract function *SyncRepo* finds the latest PushTransaction, emits a SyncPipeline event with the repo name, URL and commit hash and returns a success message. Each node's client listens for this event and performs git clone –mirror <repo-url> to reset its local state.

```
gitTransfer -push -repo "$REPO_NAME" -hash "$COMMIT_HASH" -url "$REMOTE_URL"
```

Listing 2. Post-push Hook Snippet

Model 2: Differential patch Synchronization- Here, each commit still invokes the usual post-commit hook (Listing 1) to record commit metadata. When the developer runs `git push`, the post-push hook (Listing 3) generates a patch via `git format-patch`, computes its SHA-256 hash, and submits a `RecordDiff` transaction:

```
# === Create Git Diff File ===
DIFF_DEST_DIR=~/project/S32K144_Project
DIFF_FILE="${DIFF_DEST_DIR}/${REPO_NAME}_${COMMIT_HASH}.patch"
git format-patch -1 HEAD --stdout > $DIFF_FILE
# === Sign the Diff File ===
DIFF_HASH=$(sha256sum $DIFF_FILE | awk '{print $1}')
# Record the diff on-chain
gitTransfer -recordDiff -repo "$REPO_NAME" -hash "$COMMIT_HASH" -diffhash "$DIFF_HASH" -timestamp "
    ↪ $TIMESTAMP"
```

Listing 3. Post-push Hook Snippet for Diff

The chaincode function `RecordDiff` stores `repository`, `commitHash`, `diffHash` and `timestamp`, then emits a `SyncDiffPipeline` event. Each peer's client listens for this event, fetches the patch and runs `git apply` "*reponame − commithash.patch*".

Model 3: Continuous Direct-push- In this model[6], every single commit is pushed through the pipeline immediately and drives an on-the-fly update on all peers. The post-commit hook (Listing 4) first records the commit on-chain and then invokes a Jenkins job to orchestrate the real-time synchronization. Upon receiving the push transaction highlighted in Listing 2, it uses *HandleGitPush* function.

```
# Record the commit on-chain
gitTransfer -create -hash "$COMMIT_HASH" -repo "$REPO_NAME" -message "$COMMIT_MESSAGE" -author "
    ↪ $AUTHOR"
# Trigger Jenkins pipeline for this commit
curl -X POST "http://<JENKINS_URL>/job/SyncCommitPipeline/buildWithParameters?\
token=<TOKEN>&REPO_NAME=${REPO_NAME}&COMMIT_HASH=${COMMIT_HASH}" \
  --user "<USER>:<API_TOKEN>"
```

Listing 4. Post-commit Hook Snippet for Commit Streaming

7.2 Metrics Collected

Each peer runs Prometheus Node Exporter; Prometheus scrapes every 15 s; Grafana supports live and post-run analysis. We used standard Prometheus queries against the `node_cpu_seconds_total`, `node_memory_*`, `node_filesystem_*`, *and* `node_network_receive_bytes_total` *metrics (see online supplementary material for full queries [19]).* This continuous monitoring captures synchronization latency alongside CPU utilization, memory footprint, disk consumption, and network throughput before, during, and after each Git operation. We collect the following metrics for each commit event:

(1) **Synchronization Time** measures the elapsed time from the Git push command on the originating node to the moment all peer nodes report the new commit hash in their local clone.
(2) **CPU Usage** records the average and peak processor utilization on each peer during synchronization. High CPU usage indicates resource contention or potential overload, while low usage suggests underutilized capacity.
(3) **Memory Usage** captures the resident memory consumed by the Fabric peer process and the client application. It represents the amount of main memory (RAM)[7] actively consumed by system processes, calculated as the difference between total and available memory.
(4) **Disk Usage** tracks additional storage consumed by Git operations and blockchain ledger growth. It measures the percentage of storage space consumed on the root partition[8] (`/`), highlighting available and utilized storage capacity.
(5) **Network Traffic** measures the total bytes transmitted and received per node over the network interface during synchronization. It measures the inbound data transfer rates across network interfaces.

These metrics provide a comprehensive view of performance overhead and resource demands introduced by each synchronization strategy.

7.3 Results

This subsection reports the empirical performance of the three Git–Blockchain synchronization models. We first provide a statistical overview of synchronization time and system resources across 90 runs per model, establishing baseline behavior. We then examine metric interdependence via correlation analysis, followed by pairwise significance tests with Holm–Bonferroni correction and corresponding effect sizes to assess practical

[6]It is also known as Real-Time Streaming Model.
[7]Random Access Memory
[8]The root partition ("/") is the primary filesystem directory containing essential system files and directories required for system operation.

Table 3. Comparison of Synchronization Metrics Across Models

Metric	Definition	Unit	Model 1	Model 2	Model 3
Sync Frequency	How often the repository is synchronized	Events per run	Once at the end	per push	real-time per commit
Sync Method	Git-to-blockchain integration technique	—	Full `git clone`	`git diff` + patch apply	Automated `git push` stream
Average Sync Time (Total)	Sum of all sync time	Seconds	343.44	566.62	565.1
Average Sync Time/runs	Total sync time divided by number of runs	Seconds	3.82(SD= 1.93)	6.30(SD= 4.97)	6.28(SD= 3.23)
Data Transferred	Aggregate network traffic during sync	KB/s	10.17(SD =5.66)	16.16(SD =14.05)	12.70(SD =12.13)
CPU Usage on Nodes	Peak CPU utilization observed during sync	%	8.25(SD =0.09)	7.73(SD =7.73)	8.11(SD =0.08)
Memory Usage on Nodes	Peak resident memory during sync	MB	2458 (SD =547)	2739 (SD = 537)	2432 (SD =952)
Disk Usage on Nodes	Additional storage consumed by the repo clone/patch process	GB	39.32(SD=0.78)	40.49 (SD=0.58)	40.20 (SD=0.74)
Blockchain Write (Total)	Number of ledger entries made during sync	Count	1	3	1
Blockchain Write Size (Total)	Approximate on-chain data volume	Qualitative	Low	Medium	Medium–High
Consistency Success Rate	Runs with matching commit hashes across all nodes	%	100%	100%	100%
Ease of DevOps	Subjective rating of operational complexity	1 (hard) – 5 (easy)	4	2	3
Audit Granularity	Level of forensic detail available per commit	Low / Med / High	Low	Medium	High

importance. Finally, a multiple regression identifies which resources most strongly predict synchronization latency, informing design trade-offs for deployments.

7.3.1 Statistical Overview of Synchronization and Resource Utilization. Table 3 summarizes operational characteristics across models. Model 1 performs a single complete clone at the end of development, Model 2 applies incremental patches after each push, and Model 3 streams every commit in real time. Audit granularity improves progressively from Model 1 through Model 3, while blockchain write volume grows correspondingly.

Table 4 presents descriptive statistics across 90 runs per model. Model 1 demonstrated the lowest mean synchronization time ($M = 3.82\,s, SD = 1.93$) and disk usage ($M = 39.32\,GB, SD = 0.78$), while CPU utilization remained consistent across all models ($M \approx 8\%, SD < 0.1$). Memory consumption and network throughput exhibited moderate variability. Subsequent inferential analyses examine whether these observed differences are statistically significant and practically meaningful.

Table 4. System Resource Usage and Synchronization Time Across Models(Mean (SD)

Model	Sync Time (s)	CPU (%)	Memory (MB)	Disk (GB)	Network (KB/s)
Model 1	3.82 (SD = 1.93)	8.25 (SD = 0.09)	2458.62 (SD = 546.56)	39.32 (SD = 0.78)	10.17 (SD = 5.66)
Model 2	6.30 (SD = 4.97)	7.73 (SD = 7.73)	2739.20 (SD = 537.40)	40.49 (SD = 0.58)	16.16 (SD =14.05)
Model 3	6.28 (SD = 3.23)	8.11 (SD = 0.08)	2432.26 (SD = 952.49)	40.20 (SD = 0.74)	12.70 (SD = 12.13)

7.3.2 Metric Interdependence. This section examines the interrelationships among synchronization time, CPU, memory, disk, and network usage across the three Git–blockchain synchronization models. Figure 8 presents correlation heatmaps, with Pearson's r values ranging from −1.0 to +1.0. These patterns reflect the mechanisms by which repository state is updated in each model. The most consistent feature is the strong positive correlation between disk writes and memory usage, ranging from $r = 0.45$ in Model 1 to $r = 0.88$ in Model 3.

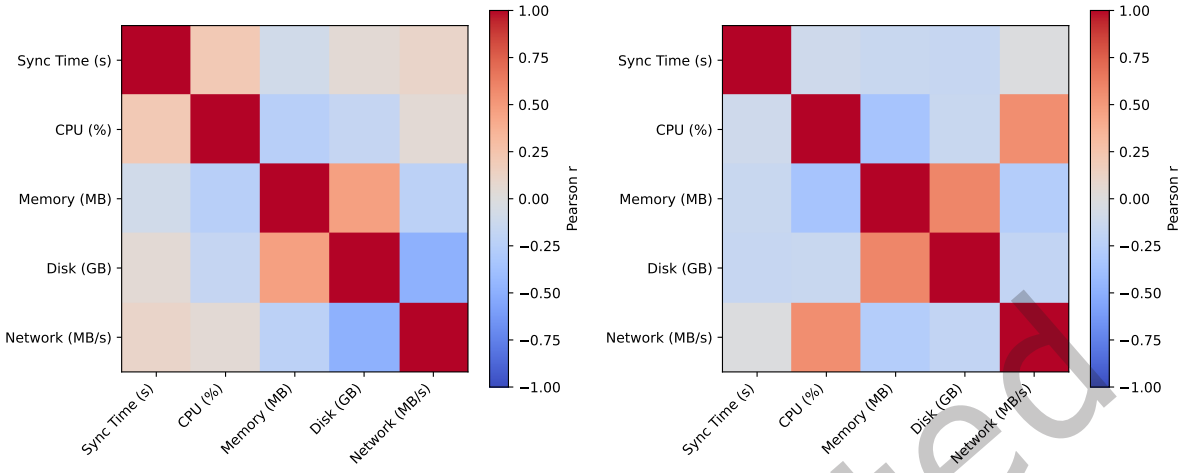
Model 1 Correlation Analysis: Memory usage and disk consumption show a moderate positive relationship ($r = 0.45$, Figure 8a). CPU load remains largely decoupled from both synchronization time ($r = -0.08$) and network throughput ($r = 0.12$). Similarly, synchronization time shows a weak negative association with memory ($r = -0.10$) and negligible correlation with network activity ($r = 0.05$). Notably, disk usage correlates weakly with synchronization time ($r = 0.15$), suggesting that storage overhead in this model has minimal direct impact on latency. These patterns indicate that the main performance bottlenecks cluster around storage and memory subsystems, while CPU and network remain comparatively stable.

Model 2 Correlation Analysis: The correlation between memory and disk strengthens ($r = 0.55$, Figure 8b). Network traffic and CPU usage show a moderate positive correlation ($r = 0.45$), indicating that heavier patch uploads and verification routines modestly stress the processor. Synchronization time remains largely decoupled from CPU ($r = -0.12$) and network ($r = 0.18$), but exhibits a slight inverse relationship with memory ($r = -0.15$). The disk–synchronization time correlation remains modest ($r = 0.22$), though stronger than in Model 1, reflecting the cumulative effect of patch application overhead. Nodes with larger memory reserves apply patches faster, reducing end-to-end latency.

Model 3 Correlation Analysis: In the continuous streaming model, memory and disk usage are highly correlated ($r \approx 0.88$ in Figure 8c), reflecting constant write operations as commits are broadcast to the ledger. Network traffic correlates substantially with CPU load ($r \approx 0.40$), consistent with frequent transaction packaging and broadcast overheads. Synchronization time shows a weak negative association with memory ($r \approx -0.20$), suggesting that additional memory buffers allow faster commit processing. Disk usage shows only a mild positive link with synchronization time ($r \approx 0.10$), implying that storage overhead introduces minimal additional latency in this streaming architecture. This profile underscores memory and disk throughput as primary levers for real-time efficiency, while network–CPU tuning delivers incremental improvements.

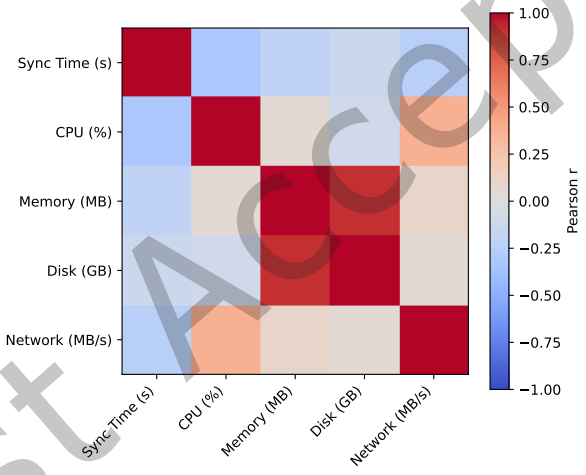
Across all three models, the memory–disk interdependence emerges as the dominant trend. CPU–network coupling is negligible in independent cloning, but noticable for diff-based and streaming models.

7.3.3 Pairwise Significance Tests. Using two-tailed Z-tests with Holm–Bonferroni correction for multiple comparisons, we compared models pairwise across the metric. Table 5 reports raw and adjusted p-values alongside Z-statistics. After controlling for family-wise error rate, seven of fifteen comparisons remained statistically significant ($p_{\text{adj}} < 0.05$), indicating performance differences across synchronization strategies. Given the large sample size (n = 90 per model), we employed two-tailed Z-tests rather than t-tests, as the sampling distribution of the mean approximates normality for samples exceeding n = 30.



(a) Correlation Matrix for Model 1

(b) Correlation Matrix for Model 2



(c) Correlation Matrix for Model 3

Fig. 8. Correlation heatmaps across three synchronization models. Each panel shows pairwise relationships between Sync Time, CPU, Memory, Disk, and Network usage.

These findings confirm that Model 1's performance advantages in synchronization latency, storage efficiency, memory footprint, and network bandwidth are genuine effects rather than statistical artifacts of inflated Type I error. The absence of CPU differences suggests that computational overhead remains constant regardless of synchronization strategy, while differences in storage and data transfer operations explain the observed performance variations.

Table 5. Pairwise Two-Tailed Z-Test Results with Holm–Bonferroni Correction ($\alpha = 0.05$, 15 comparisons)

Metric	Comparison	Raw p-value	Adjusted p†	Z-statistic	Significant?
Sync Time (s)	M1 vs M2	< 0.001	< 0.001	-4.41	Yes**
	M1 vs M3	< 0.001	< 0.001	-6.21	Yes**
	M2 vs M3	0.978	0.978	0.03	No
CPU (%)	M1 vs M2	0.710	1.000	0.37	No
	M1 vs M3	0.913	1.000	0.11	No
	M2 vs M3	0.795	1.000	-0.26	No
Memory (MB)	M1 vs M2	< 0.001	0.005	-3.49	Yes**
	M1 vs M3	0.829	1.000	0.22	No
	M2 vs M3	0.008	0.062	2.66	No
Disk (GB)	M1 vs M2	< 0.001	< 0.001	-11.46	Yes**
	M1 vs M3	< 0.001	< 0.001	-7.77	Yes**
	M2 vs M3	0.004	0.032	2.91	Yes**
Network (KB/s)	M1 vs M2	< 0.001	0.002	-3.75	Yes**
	M1 vs M3	0.073	0.511	-1.79	No
	M2 vs M3	0.077	0.461	1.77	No

7.3.4 Effect Size Analysis. While statistical significance indicates whether observed differences are likely genuine rather than due to chance, effect size quantifies the *magnitude* of those differences in standardized units, enabling assessment of practical importance [22]. We report Cohen's d for all seven statistically significant comparisons identified in Table 5, calculated as:

$$d = \frac{M_1 - M_2}{\sqrt{\frac{(n_1-1)SD_1^2+(n_2-1)SD_2^2}{n_1+n_2-2}}} \tag{1}$$

where M_1 and M_2 are group means, SD_1 and SD_2 are standard deviations, and n_1 and n_2 are sample sizes (both equal to 90). Cohen's d values are typically interpreted as small ($d \approx 0.2$), medium ($d \approx 0.5$), or large ($d \approx 0.8$). Table 6 summarizes effect sizes for significant comparisons.

Synchronization Time: Model 1's latency advantage over Model 2 represents a medium effect, while the advantage over Model 3 constitutes a large effect. These effect sizes translate to approximately 2.5-second reductions.

Disk Usage: Model 1's storage efficiency yields exceptionally large effects compared to both Model 2 and Model 3. Although reductions of 0.88–1.17 GB may appear modest, they represent 2.2–2.9% decreases that compound over repeated synchronization cycles. For *Model 2 vs. Model 3*, The effect indicates that Model 2 consumes moderately more disk space than Model 3. While this difference is less pronounced than Model 1's advantages, it suggests that incremental commit streaming (Model 3) manages storage more efficiently than accumulating patch files (Model 2).

Table 6. Effect Sizes (Cohen's *d*) for Statistically Significant Comparisons

Metric	Comparison	Cohen's *d*	95% CI	Interpretation
Sync Time	M1 vs. M2	-0.66	[-0.96, -0.36]	Medium
Sync Time	M1 vs. M3	-0.93	[-1.23, -0.62]	Large
Disk	M1 vs. M2	-1.70	[-2.04, -1.36]	Large
Disk	M1 vs. M3	-1.16	[-1.47, -0.84]	Large
Disk	M2 vs. M3	0.44	[0.14, 0.73]	Small
Memory	M1 vs. M2	-0.52	[-0.81, -0.22]	Medium
Network	M1 vs. M2	-0.56	[-0.86, -0.26]	Medium

Memory and Network: Both comparisons exhibit medium effects. Model 1's 282 MB memory reduction (10.3%) and 6 KB/s network bandwidth savings (37%) are practically significant in resource-constrained CI/CD environments where concurrent pipelines compete for shared infrastructure.
These effect sizes confirm that Model 1's statistical superiority. The large effects for disk usage and synchronization time justify prioritizing Model 1 in latency-sensitive or storage-constrained deployments. However, the trade-off must be weighed against organizational requirements.

7.3.5 Regression Analysis. To identify which system resources most strongly influence synchronization latency, we fitted a multiple linear regression model with synchronization time as the dependent variable and CPU utilization, memory footprint, disk consumption, and network throughput as independent predictors. The model is summarized in Table 7 summarizes the model results. Negligible ↓/↑ indicates that although the coefficient points in one direction, the effect is statistically non-significant and close to zero.

Table 7. Multiple Linear Regression Predicting Synchronization Time

Predictor	Coefficient	Std. Error	t-Statistic	p-Value	Effect on Sync Time
Intercept	2.15	0.48	4.48	<0.001	Baseline (2.15s)
CPU	-0.02	0.03	-0.67	0.503	Negligible ↓
Memory	-0.0013	0.0004	-3.25	0.0013	Small ↓ (−1.3ms per MB)
Disk	+1.12	0.12	9.33	<0.001	Large ↑ (1.12s per GB)
Network	+0.004	0.006	0.67	0.503	Negligible ↑

Memory and disk usage emerged as the only statistically significant predictors. Specifically, each additional gigabyte of disk consumption increases synchronization time by 1.12 seconds, highlighting the high cost of storage I/O during repeated clone and write operations. Conversely, each additional megabyte of memory decreases latency by 0.0013 seconds.

The regression confirms that minimizing disk writes is the most effective strategy for reducing synchronization time. The disk coefficient (+1.12 s/GB) quantifies this universal storage bottleneck and explains Model 1's observed 2.4–2.5 second latency advantage. Its lower disk footprint (39.32 GB vs. 40+ GB for Models 2 and 3) directly translates into faster synchronization. Model 1 achieves this by incurring the highest absolute disk overhead

through independent cloning. Model 3 balances modest disk footprint with efficient memory use and automation to yield consistent performance without sacrificing auditability.

8 Discussion

Our evaluation of the three Git–blockchain synchronization models reveals a nuanced performance landscape where Model 1 demonstrates statistically significant advantages in synchronization latency and resource efficiency, yet practical deployment decisions must account for scalability constraints, audit requirements, and infrastructure characteristics that extend beyond raw performance metrics.

8.1 Synthesis of Statistical Findings

Four complementary analytical approaches (descriptive statistics, correlation analysis, pairwise significance tests, and multiple regression) converge on a consistent narrative that disk I/O dominates synchronization performance across all three models, while CPU and network resources remain underutilized.

Model 1 outperformed Models 2 and 3 on synchronization time (39.4% and 39.2% reductions, respectively), disk usage (1.17 GB and 0.88 GB savings), memory consumption (281 MB reduction vs. Model 2), and network throughput (5.99 KB/s reduction vs. Model 2). CPU utilization showed no significant differences across any model pair, indicating that computational overhead remains constant regardless of synchronization strategy.

Effect size analysis reveals that these statistically significant differences translate into practically meaningful impacts. The large effect sizes for disk usage (Cohen's $d = -1.70$ for Model 1 vs. Model 2; $d = -1.16$ for Model 1 vs. Model 3) are rare in applied research and indicate substantive real-world differences. Synchronization time effects ranged from medium ($d = -0.66$) to large ($d = -0.93$), while memory and network effects were consistently medium ($d \approx -0.52$ to -0.56). Effect sizes exceeding 1.0 standard deviations suggest that Model 1's disk efficiency advantage. It reflects a fundamental architectural difference in how repository state is managed.

Section 5.1 presented three hypotheses: H1 (synchronization strategy significantly affects latency), H2 (synchronization strategy significantly affects resource usage), and H3 (resource metrics significantly influence synchronization time). Our statistical analyses support all three hypotheses. Pairwise Z-tests with Holm–Bonferroni correction revealed significant differences in synchronization time between Model 1 and the other two models ($p < 0.01$), confirming H1. Significant differences were also observed in disk usage, memory consumption and network throughput, supporting H2. Multiple regression analysis showed that disk usage is a strong predictor of synchronization time ($\beta \approx +1.12 s/GB$, $p < 0.001$) and memory usage is inversely associated with latency ($\beta \approx -0.0013 s/MB$, $p = 0.001$), supporting H3. CPU utilization and network throughput did not significantly predict synchronization time under the experimental conditions.

In the regression analysis, disk consumption emerged as the dominant latency predictor ($\beta = +1.12$ s/GB, $p < 0.001$), indicating that each additional gigabyte of storage overhead adds 1.12 seconds to synchronization time. Memory consumption showed a modest inverse relationship ($\beta = -0.0013$ s/MB, $p = 0.001$), suggesting that larger memory buffers enable faster processing through reduced disk swapping. Neither CPU utilization nor network throughput predicted synchronization time, confirming that these resources were not bottlenecks under our experimental conditions. These findings illuminate an apparent paradox in the pairwise results: although Model 1 transferred significantly less network data than Model 2 (10.17 KB/s vs. 16.16 KB/s, $p_{\text{adj}} = 0.002$), network throughput did not predict latency. The network differences does not systematically drive synchronization time because bandwidth was sufficient for all three approaches. The performance separation stems instead from disk I/O efficiency, where Model 1's lower footprint (39.32 GB vs. 40+ GB) directly translates into its observed 2.4–2.5 second latency advantage via the regression coefficient.

Correlation analysis (Section 7.3.2) exposes the interdependencies that propagate disk bottlenecks throughout the system. The memory-disk correlation intensifies across model reflecting the reality that repository updates

must be staged in memory before being flushed to persistent storage. This has direct implications for capacity planning as provisioning additional memory alone will not improve performance if disk I/O remains saturated. The CPU-network coupling observed in Models 2 and 3 but absent in Model 1 reveals how patch-based and streaming architectures introduce computational overhead for diff generation, validation, and transmission. Yet despite this coupling, CPU utilization never reaches levels that impact synchronization time.

8.2 Interpretation

Model 1's performance advantages stem from its architectural simplicity and deferred synchronization strategy. By postponing repository updates until push events and executing a single full clone, Model 1 minimizes blockchain transaction overhead (one ledger write per push vs. three for Model 2 or continuous writes for Model 3) and consolidates disk I/O into discrete, optimizable operations. The full clone strategy, while seemingly wasteful, benefits from Git's internal optimizations for bulk transfers. Additionally, Model 1's lower memory consumption reflects its stateless operation.

Model 2's elevated memory consumption (2739 MB vs. 2458 MB for Model 1, $d = -0.52$) stems from the buffering requirements of differential patch synchronization. Each `git format-patch` operation loads commit objects into memory, computes diffs, generates patch files, and stages them for blockchain submission. The accumulated patch files also explain Model 2's higher disk usage. Although individual patches are small (<1 MB), they persist on disk until applied. The moderate CPU-network correlation ($r = 0.45$) reflects the computational cost of patch generation and validation, yet this overhead never saturates the processor for CPU utilization remained below 8% across all models.

Model 3's continuous streaming architecture exhibits the tightest memory-disk coupling ($r = 0.88$) because every commit immediately triggers both memory allocation and disk writes (for ledger updates and local repository increments). This real-time operation explains Model 3's higher disk usage (40.20 GB) despite transmitting less data than Model 2. The blockchain ledger grows continuously rather than in batches, and small, frequent writes incur greater filesystem metadata overhead than bulk operations. The high standard deviation in memory consumption reflects the variability inherent in streaming workloads. Despite this variability, Model 3's mean synchronization time (6.28 s) differs negligibly from Model 2 (6.30 s), suggesting that fine-grained commit streaming does not inherently improve latency. The disk I/O bottleneck persists regardless of update granularity.

The absence of significant CPU differences across all model pairs is both encouraging and revealing. It indicates that Git operations and blockchain transaction processing impose minimal and equivalent computational overhead, leaving CPU capacity available for concurrent development tasks. However, this finding is workload-dependent. High-frequency scenarios may reveal CPU bottlenecks as transaction signing, hashing, and consensus operations accumulate. The current results suggest that for typical development workflows, CPU provisioning is not a critical concern for Git-blockchain synchronization.

8.3 Functional Reliability

All three models achieved 100% consistency success rates(Table 3). Every synchronization event across 270 total runs (90 per model) resulted in identical commit hashes across all nodes, with zero failures, retries, or state divergences. This validates the fundamental soundness of blockchain-backed Git synchronization and confirms that blockchain provide sufficient integrity guarantees for distributed version control. The consistency achievement shifts the research question from "*Can* blockchain synchronize Git repositories reliably?" to "*Which synchronization strategy optimizes performance while maintaining reliability?*" All three models are functionally correct. The selection criteria hinge on performance trade-offs, audit granularity requirements, and operational constraints rather than correctness concerns.

8.4 Audit Granularity vs. Performance Trade-offs

The models were architected to provideauditability while guaranteeing deterministic state convergence across nodes. Model 1's coarse audit granularity—recording only push events rather than individual commits—limits forensic capabilities. In compliance-driven automotive environments where regulators may demand commit-level provenance, Model 3's real-time commit streaming provides superior traceability. While our evaluation prioritized performance metrics, regulatory or contractual requirements may mandate fine-grained auditability, making Model 3 the *de facto* choice regardless of performance penalties. Furthermore, Model 3 better aligns with modern DevOps practices where continuous integration pipelines trigger on individual commits, enabling real-time CI/CD integration and automated testing, whereas Model 1's batch-oriented synchronization introduces delays between commit creation and blockchain verification that may violate service-level agreements for build turnaround time.

Beyond quantitative performance metrics, developers must contend with ergonomic factors that influence day-to-day productivity. High-frequency commit cycles and merge-heavy workflows can introduce perceptible latency in blockchain synchronized pipelines. A stream of small commits on a monorepo may saturate the blockchain ordering service, leading to queueing delays and merge conflicts. Conversely, batching multiple commits into a single push reduces ledger writes and improves throughput but sacrifices fine-grained provenance. Large monorepos exacerbate these issues by increasing diff generation times and repository clone sizes. To mitigate these challenges, practitioners can employ lazy batching (grouping commits into configurable time windows), asynchronous commit anchoring (recording commit hashes on-chain without awaiting full endorsement before proceeding), and selective commit hashing (anchoring only release-candidate or tag commits on the blockchain while storing full diffs off-chain). These strategies balance developer ergonomics against audit requirements, reducing on-chain write volume while preserving essential provenance.

8.5 Deployment Recommendations

While Model 1 demonstrates statistical superiority across multiple metrics, interpreting these results requires balancing empirical findings against operational practicality, infrastructural constraints, and organizational priorities. Based on our findings, we propose context-dependent selection criteria:

- **High-frequency development with small repositories:** Model 1 offers optimal latency and minimal blockchain overhead, suitable for agile teams prioritizing speed over audit granularity. The latency reduction and cumulative storage savings justify adoption for organizations with aggressive development velocity and tight CI/CD turnaround requirements.
- **Large repositories with regulatory compliance requirements:** Model 3 provides fine-grained commit-level auditability and scales gracefully through incremental updates. Accept the latency penalty as the cost of forensic traceability. The strong memory-disk correlation necessitates co-provisioning both resources to avoid bottlenecks.
- **Bandwidth-constrained environments:** Model 2 minimizes network traffic (though higher than Model 1) while maintaining moderate audit detail through patch-level tracking. Suitable for geographically distributed teams with limited WAN bandwidth or high data transfer costs.
- **Enterprise data centers with NVMe storage:** Model 1's performance advantage compounds on high-speed infrastructure where disk I/O no longer dominates latency. The disk coefficient becomes negligible when storage operations complete in milliseconds, making full clones competitive even for larger repositories.
- **Memory-constrained CI/CD runners:** Model 3 or Model 1 require less RAM than Model 2. For containerized build environments with strict memory limits, avoid Model 2's patch buffering overhead.

All three models proved functionally effective, achieving 100% consistency success rates. The choice hinges on organizational priorities (latency vs. audit granularity vs. scalability) rather than fundamental correctness

concerns. Organizations should profile their specific workload characteristics against these empirical benchmarks to select the optimal synchronization strategy.

9 Limitations and Future Work

This work has several limitations that contextualize its findings and motivate future research directions. Our experiments employed virtual machines with consumer-grade hardware in a controlled network environment. While this reflects typical developer workstation configurations, enterprise infrastructure with NVMe storage and multi-core servers may exhibit different absolute latency values, though relative model rankings remain driven by algorithmic differences. The synchronization models operate exclusively within OEM/supplier development infrastructure. Vehicle ECUs do not participate in Git operations and are unaffected by the resource trade-offs discussed here. Future work should benchmark performance across infrastructure tiers (developer workstations, build servers, cloud CI/CD farms) and account for real-world conditions including intermittent connectivity, variable network latency, and organizational endorsement policies.

Our evaluation used a NXP S32K144 microcontroller project, which may not generalize to multi-gigabyte enterprise repositories with extensive binary assets and commit histories. Repositories exceeding 100 MB could shift cost balances between cloning and incremental synchronization. Future experiments should validate scalability across repository sizes from 100 MB to 1 GB to identify inflection points.

Hyperledger Fabric documentation notes that a minimum transaction size with no endorsements is around 1 KB and that a typical transaction with endorsements is approximately 3–4 KB. Therefore, each on-chain commit or patch recorded by our models adds roughly 3–4 KB to the ledger. Under Model 3's continuous streaming, a repository with 10,000 commits could expand the ledger by 30–40 MB. This growth is manageable for modern disks but becomes significant at millions of commits or when storing diffs and artifacts on-chain. Ledger replication cost scales with both the number of commits and the number of organizations: a 50-organization consortium would collectively replicate an additional 1.5–2.0 GB of ledger data for 50,000 commits, assuming 3–4 KB per commit. Practitioners should plan storage and backup accordingly. Additionally, empirical studies have shown that Hyperledger Fabric networks may experience failures when more than 1000 transactions are sent simultaneously due to process concurrency and communication overhead. Although our workloads remained well below this threshold, deployments spanning dozens of organizations must consider batching transactions, adjusting consensus parameters and distributing ordering services geographically.

Our 5-node testbed reflects a simplified supply chain. Real-world automotive ecosystems involve 50+ organizations (OEMs, Tier 1/2 suppliers, certification authorities). While Hyperledger Fabric supports up to 20,000 transactions per second [24], scaling to large consortia introduces three challenges: (1) endorsement latency increases with geographically distributed stakeholders; (2) ledger growth becomes significant; (3) governance complexity arises when coordinating endorsement policies. Models 1 and 2, with lower blockchain write frequencies, may scale more gracefully. Future work should validate these projections using multi-organizational testbeds (10–20 nodes) and explore sharding or layer-2 strategies.

This study did not incorporate adversarial scenarios such as targeted packet loss, ordering node censorship, malicious chaincode deployment, Byzantine faults, Sybil attacks, or timing exploits. Empirical security validation under hostile conditions would strengthen claims about tamper resistance and resilience.

Future research should explore dynamic synchronization strategies that adapt between full clones, differential patches, and streaming based on repository size, network conditions, and commit frequency. Such hybrid approaches could balance storage overhead against latency more effectively than static model selection.

Another important next step is to leverage Git–blockchain triggers to initiate reproducible builds across distributed nodes, thereby creating a seamless pipeline from code commit through build completion. Building on these directions, future work will focus on deploying our proposed **GuixChain framework**—an end-to-end

supply chain security platform integrating Git, blockchain, reproducible builds, and Software Bills of Materials (SBOMs) into a unified provenance system. Our vision for GuixChain extends beyond existing reproducible build frameworks such as Nix and Guix, which provide deterministic builds but lack blockchain-backed provenance, or in-toto, which offers supply chain attestations but does not embed reproducible build semantics into OTA pipelines.

GuixChain extends our work by: (1) triggering deterministic builds automatically upon commit verification, eliminating nondeterministic compiler behavior and timestamp variations; (2) extracting dependency graphs and recording them immutably on the ledger to detect supply chain substitution attacks; and (3) delivering signed binaries to vehicle ECUs via Uptane [32], with blockchain providing the root-of-trust for image repository metadata and time-server attestations.

Embedding reproducible build workflows and SBOM generation directly into the blockchain ledger [5], GuixChain provides comprehensive provenance guarantees, extending trust and traceability from source code through deployed binary integrity. This vision transforms blockchain from a passive audit log into an active orchestrator of secure software delivery, bridging source code provenance with deployed binary integrity and addressing both the technical and operational requirements of IoV ecosystems, ensuring resilience, transparency, and accountability.

10 Conclusion

This paper presents the first controlled, quantitative comparison of three Git–blockchain synchronization models for automotive SSCs. Through 270 experimental runs (90 per model), we evaluated synchronization latency, resource consumption, and system scalability using rigorous statistical methods including pairwise Z-tests with Holm–Bonferroni correction, effect size analysis, correlation analysis, and multiple regression. Our analysis showed that disk footprint is the dominant driver of synchronization time, with Model 1 achieving consistent advantages in efficiency, while Models 2 and 3 offer viable trade-offs for bandwidth-constrained or compliance-driven environments. All three models achieved 100% consistency, demonstrating the feasibility of integrating version control with blockchain to enforce tamper-evident provenance.

Our findings reveal that Model 1 (independent clone) achieves statistically superior performance across four of five metrics. The findings highlight that synchronization choices cannot be reduced to raw performance metrics alone. In latency-sensitive OTA pipelines, Model 1 variants are most effective; in multi-stakeholder environments requiring detailed traceability, Model 3 is preferable; and in resource-constrained deployments, Model 2 balances overhead with efficiency. These insights provide actionable guidance for OEMs and suppliers seeking to harden software pipelines against compromise.

Beyond performance, this work establishes foundational principles for integrating version control with distributed ledgers in safety-critical domains. Blockchain provides immutability, decentralized verification, and non-repudiation, which extend Git's versioning into a robust provenance framework. The integration transforms passive audit logs into active security mechanisms that detect tampering, attribute actions to specific actors, and provide forensic evidence for incident investigation. This shifts SSCs from trust-based coordination to cryptographically verifiable collaboration.

Future work will extend these synchronization primitives into GuixChain, an end-to-end supply chain security platform integrating git, blockchain, reproducible builds, SBOM, and secure OTA deployment. By embedding deterministic build processes and on-chain provenance capture, GuixChain will bridge source code commits to deployed binary integrity. Additional research should validate performance across infrastructure tiers (developer workstations, cloud CI/CD farms), benchmark scalability with multi-gigabyte repositories, conduct adversarial evaluation under Byzantine fault conditions, and explore hybrid strategies that dynamically adapt synchronization methods based on repository size and network conditions.

This research contributes to the design of trustworthy, transparent, and continuously verifiable software supply chains for connected vehicles, offering both practical recommendations and a foundation for next-generation supply chain security.

Acknowledgment: This work was supported by Clemson University's Virtual Prototyping of Autonomy Enabled Ground Systems (VIPR-GS), under Cooperative Agreement W56HZV-21-2-0001 with the US Army DEVCOM Ground Vehicle Systems Center (GVSC).

References

[1] 2016. Public versus Private Blockchains Part 1 : Permissioned Blockchains. https://api.semanticscholar.org/CorpusID:44197419

[2] 2021. SolarWinds Cyberattack Demands Significant Federal and Private-Sector Response (Infographic). https://www.gao.gov/blog/solarwinds-cyberattack-demands-significant-federal-and-private-sector-respons [Online; posted on April 22, 2021].

[3] Wafaa AH Ahmed, Bart L MacCarthy, and Horst Treiblmaier. 2022. Why, where and how are organizations using blockchain in their supply chains? Motivations, application areas and contingency factors. *International Journal of Operations & Production Management* 42, 12 (2022), 1995–2028.

[4] W. A. H. Ahmed, B. L. MacCarthy, and H. Treiblmaier. 2022. Why, Where and How Are Organizations Using Blockchain in Their Supply Chains? Motivations, Application Areas and Contingency Factors. *International Journal of Operations & Production Management* 42, 12 (2022), 1995–2028. doi:10.1108/IJOPM-12-2021-0805

[5] Iwinosa Aideyan, Richard Brooks, and Mert Pesé. 2025. Supply chain forensics with distributed ledger technologies, software bill of materials, and AI-enabled forensic investigations. In *Proceedings of the Network in Digital Sciences Conference*. Springer Nature, Cham, Switzerland. Accepted for publication.

[6] Iwinosa W. Aideyan. 2025. Blockchain-Integrated Version Control for Secure and Transparent Software Supply Chains. https://open.clemson.edu/all_theses/4524 All Theses, Paper 4524.

[7] Amina Y. Alsallut, Ruba A. Salamah, and Aiman A. Abusamra. 2023. Exploratory Study on Hyperledger Fabric Framework: Food Supply Chain as a Case Study. *International Journal of Engineering and Manufacturing* (2023). doi:10.5815/ijem.2023.04.02

[8] P A Andreev and E Krotova. 2018. Review of Blockchain Technology: Types of Blockchain and Their Application. *Intellekt. Sist. Proizv.* 16, 1 (Apr. 2018), 11–14. doi:10.22213/2410-9304-2018-1-11-14

[9] P. Baglietto, M. Maresca, Andrea Parodi, and Davide Senatori. 2021. Application of the Hyperledger Fabric Blockchain in a supply and maintenance chain for critical assets. *The 23rd International Conference on Information Integration and Web Intelligence* (2021). doi:10.1145/3487664.3487752

[10] Zbigniew Banach. 2024. *The xz-utils Backdoor: The Supply Chain RCE That Got Caught.* https://www.invicti.com/blog/web-security/xz-utils-backdoor-supply-chain-rce-that-got-caught/ Accessed October 27, 2025.

[11] Shehar Bano, Alberto Sonnino, Mustafa Al-Bassam, Sarah Azouvi, Patrick McCorry, Sarah Meiklejohn, and George Danezis. 2019. SoK: Consensus in the age of blockchains. In *Proceedings of the 1st ACM Conference on Advances in Financial Technologies*. 183–198.

[12] Jon Boyens, Robert McWhite, Lisa Calloway, Nadya Bartol, and Karen Scarfone. 2024. *NIST Cybersecurity Framework 2.0: Quick-Start Guide for Cybersecurity Supply Chain Risk Management (C-SCRM).* Special Publication (NIST SP 1305). National Institute of Standards and Technology (NIST), Gaithersburg, MD. doi:10.6028/NIST.SP.1305 Accessed October 27, 2025.

[13] Richard R. Brooks, Lu Yu, and Anthony Skjellum. 2023. Management of Access Authorization Using an Immutable Ledger. https://patents.google.com/patent/US11646872B2/en Agency: Kim and Lahey Law Firm, LLC.

[14] Prithwiraj Choudhury, Kevin Crowston, Linus Dahlander, Marco S Minervini, and Sumita Raghuram. 2020. GitLab: work where you want, when you want. *Journal of Organization Design* 9, 1 (2020), 23.

[15] Ian Clatworthy. 2007. Distributed Version Control Systems Why and How. https://api.semanticscholar.org/CorpusID:115072

[16] CMSTAT. 2020. Configuration Management Across Multi-Site Aerospace & Defense Suppliers – Part 1. https://cmstat.com/cmsights-news-posts/configuration-management-across-multi-site-aerospace-defense-suppliers-part-1

[17] Valerio Cosentino, Javier L Cánovas Izquierdo, and Jordi Cabot. 2017. A systematic mapping study of software development with GitHub. *Ieee access* 5 (2017), 7173–7192.

[18] Brian De Alwis and Jonathan Sillito. 2009. Why are software projects moving from centralized to decentralized version control systems?. In *2009 ICSE Workshop on Cooperative and Human Aspects on Software Engineering*. IEEE, 36–39.

[19] Prometheus Documentation. n.d.. *Prometheus Documentation.* https://prometheus.io/docs/ Accessed: March 4, 2025.

[20] Pedro Ferreira, Filipe Caldeira, Pedro Martins, and Maryam Abbasi. 2023. Log4j Vulnerability. In *International Conference on Information Technology & Systems*. Springer, 375–385.

[21] GeeksforGeeks. 2024. *Git – Hooks.* https://www.geeksforgeeks.org/git-hooks/ Last Updated: June 30, 2024. Accessed: March 4, 2025.

[22] Gilles E Gignac and Eva T Szodorai. 2016. Effect size guidelines for individual differences researchers. *Personality and individual differences* 102 (2016), 74–78.

[23] Google Cloud. 2018. Grafeas: An open artifact metadata API. https://grafeas.io.
[24] Christian Gorenflo, Stephen Lee, Lukasz Golab, and Srinivasan Keshav. 2020. FastFabric: Scaling hyperledger fabric to 20 000 transactions per second. *International Journal of Network Management* 30, 5 (2020), e2099.
[25] Luca Grilli and Paolo Speziali. 2024. Combining Git and Blockchain for Trusted Information Sharing. *IEEE Access* (2024).
[26] Muhammad Hammad, Jawaid Iqbal, Saddam Hussain, Syed Sajid Ullah, Mueen Uddin, Urooj Ali Malik, Maha Abdelhaq, Raed Alsaqour, et al. 2023. Blockchain-based decentralized architecture for software version control. *Applied Sciences* 13, 5 (2023), 3066.
[27] Badis Hammi and Sherali Zeadally. 2023. Software supply-chain security: Issues and countermeasures. *Computer* 56, 7 (2023), 54–66.
[28] Alan R Hevner, Salvatore T March, Jinsoo Park, and Sudha Ram. 2004. Design science in information systems research. *MIS quarterly* (2004), 75–105.
[29] Hyperledger Fabric Documentation. 2025. Introduction to Hyperledger Fabric. https://hyperledger-fabric.readthedocs.io/en/release-2.5/whatis.html. [Online; accessed February 2025].
[30] Nallam Sri Venkata Kalyan et al. 2025. Blockchain-Enabled DevSecOps Pipeline for Automated Compliance and Security Audits. *Journal of DevOps Security* (2025).
[31] Sachin S Kamble, Angappa Gunasekaran, Nachiappan Subramanian, Abhijeet Ghadge, Amine Belhadi, and Mani Venkatesh. 2023. Blockchain technology's impact on supply chain integration and sustainable supply chain performance: Evidence from the automotive industry. *Annals of Operations Research* 327, 1 (2023), 575–600.
[32] Trishank Karthik, Akan Brown, Sebastien Awwad, Damon McCoy, Russ Bielawski, Cameron Mott, Sam Lauzon, André Weimerskirch, and Justin Cappos. 2016. Uptane: Securing software updates for automobiles. In *International conference on embedded security in car*, Vol. 11.
[33] Mahtab Kouhizadeh, Sara Saberi, and Joseph Sarkis. 2021. Blockchain technology and the sustainable supply chain: Theoretically exploring adoption barriers. *International journal of production economics* 231 (2021), 107831.
[34] Grafana Labs. n.d.. *Grafana Documentation.* https://grafana.com/docs/ Accessed: March 4, 2025.
[35] Linux Foundation. 2021. Sigstore: A new standard for signing, verifying and protecting software. https://sigstore.dev.
[36] Jeferson J.F. Martinez and Javier M. Durán. 2021. Software Supply Chain Attacks, a Threat to Global Cybersecurity: SolarWinds' Case Study. *International Journal of Safety and Security Engineering* (2021). https://api.semanticscholar.org/CorpusID:244056951
[37] Robert Mitchell and Ing-Ray Chen. 2014. A survey of intrusion detection techniques for cyber-physical systems. *ACM Computing Surveys (CSUR)* 46, 4 (2014), 1–29.
[38] Chandrasekaran Mohan. 2019. State of Public and Private Blockchains: Myths and Reality. *Proceedings of the 2019 International Conference on Management of Data* (2019). doi:10.1145/3299869.3314116
[39] JOSÉ MIGUEL PEREIRA MURTA. February,2024. *TRUSTWORTHY CLASSIC CAR RESTORATION'S HISTORY USING BLOCKCHAIN.* Master's thesis. NOVA University Lisbon.
[40] Rana M. Nadir. 2019. Comparative study of permissioned blockchain solutions for enterprises. *2019 International Conference on Innovative Computing (ICIC)* (2019), 1–6. https://api.semanticscholar.org/CorpusID:210931885
[41] Arvind Narayanan, Joseph Bonneau, Edward Felten, Andrew Miller, and Steven Goldfeder. 2016. *Bitcoin and Cryptocurrency Technologies: A Comprehensive Introduction.* Princeton University Press, USA.
[42] Qassim Nasir, Ilham A Qasse, Manar Abu Talib, and Ali Bou Nassif. 2018. Performance analysis of hyperledger fabric platforms. *Security and Communication Networks* 2018, 1 (2018), 3976093.
[43] N. Nizamuddin, K. Salah, M. Ajmal Azad, J. Arshad, and M.H. Rehman. 2019. Decentralized document version control using ethereum blockchain and IPFS. *Computers & Electrical Engineering* 76 (2019), 183–197. doi:10.1016/j.compeleceng.2019.03.014
[44] Nadav Noy. 2022. *Toyota Customer Data Leaked Due to Software Supply Chain Attack.* https://www.legitsecurity.com/blog/toyota-customer-data-leaked-due-to-software-supply-chain-attack Accessed October 27, 2025.
[45] Eric O'Donoghue, Ann Marie Reinhold, and Clemente Izurieta. 2024. Assessing Security Risks of Software Supply Chains Using Software Bill of Materials. *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering - Companion (SANER-C)* (2024), 134–140. https://api.semanticscholar.org/CorpusID:267336705
[46] OX Security. 2024. Supply Chain Security for Software: What You Need to Know. https://www.ox.security/supply-chain-security-for-software-what-you-need-to-know/. [Online; published 20-Nov-2024].
[47] Guido Perboli, Stefano Musso, and Mariangela Rosano. 2018. Blockchain in logistics and supply chain: A lean approach for designing real-world use cases. *Ieee Access* 6 (2018), 62018–62028.
[48] Julien Polge, J. Robert, and Y. L. Traon. 2020. Permissioned blockchain frameworks in the industry: A comparison. *ICT Express* 7 (2020), 229–233. doi:10.1016/J.ICTE.2020.09.002
[49] R2Devops. 2024. *Top 5 Software Supply Chain Security Incidents.* https://docs.r2devops.io/blog/top-5-cybersecurity-incidents-in-cicd/ Accessed: 2025-04-12.
[50] Kotha Raj Kumar Reddy, Angappa Gunasekaran, P Kalpana, V Raja Sreedharan, and S Arvind Kumar. 2021. Developing a blockchain framework for the automotive supply chain: A systematic review. *Computers & Industrial Engineering* 157 (2021), 107334.
[51] Thomas Reeves. 2006. Design research from a technology perspective. In *Educational design research.* Routledge, 64–78.

[52] M. H. Rehmani. 2021. Blockchain Fundamentals and Working Principles. (2021), 23–59. doi:10.1007/978-3-030-71788-9_3
[53] Sabbir M. Saleh et al. 2024. Blockchain for Securing CI/CD Pipeline: A Comprehensive Review. In *Proceedings of Cloud Computing Conference*.
[54] Siemens. 2018. OEMs, Suppliers, and How to Use a Program Management System. https://blogs.sw.siemens.com/thought-leadership/2018/12/20/oems-suppliers-and-how-to-use-a-program-management-system/. Accessed: 2024-04-27.
[55] SLSA Community. 2021. SLSA: Supply chain Levels for Software Artifacts. https://slsa.dev.
[56] Nick Szabo. 1997. Formalizing and securing relationships on public networks. *First monday* (1997).
[57] Edward Targett. 2025. *Jeep Software Update Bricks Vehicles, Leaves Owners Stranded.* https://www.thestack.technology/jeep-software-update-bricks-vehicles-leaves-owners-stranded/ Accessed October 27, 2025.
[58] Git Documentation Team. 2025. *8.3 Customizing Git - Git Hooks.* https://git-scm.com/book/en/v2/Customizing-Git-Git-Hooks Accessed: March 4, 2025.
[59] Wensheng Tian, Lei Zhang, Shuangxi Chen, Hu Wang, and Xiao Luo. 2023. Poster: A Privacy-Preserving Smart Contract Vulnerability Detection Framework for Permissioned Blockchain. *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security* (2023). doi:10.1145/3576915.3624366
[60] Ulpan Tokkozhina, Ana Lucia Martins, and Joao C Ferreira. 2023. Uncovering dimensions of the impact of blockchain technology in supply chain management. *Operations Management Research* 16, 1 (2023), 99–125.
[61] Santiago Torres-Arias, Hammad Afzali, Trishank Karthik Kuppusamy, Reza Curtmola, and Justin Cappos. 2019. in-toto: Providing farm-to-table guarantees for bits and bytes. In *28th USENIX Security Symposium*. 1393–1410.
[62] Santiago Torres-Arias, Anil Kumar Ammula, Reza Curtmola, and Justin Cappos. 2016. On omitting commits and committing omissions: Preventing git metadata tampering that (re) introduces software vulnerabilities. In *25th USENIX Security Symposium (USENIX Security 16)*. 379–395.
[63] AMRUTA SUDHIR VATARE and PRATIBHA ADKAR. 2019. Review Paper on Centralized and Distributed Version Control System. (2019).
[64] VinChain. 2025. *VinChain API Documentation.* https://vinchain.io/api-docs Accessed: March 4, 2025.
[65] David A Wheeler. 2015. Secure programming HOWTO. *Walters Art Museum in Baltimore, Maryland* (2015).
[66] Gyan Wickremasinghe, S. Frost, Karen Rafferty, and Vishal Sharma. 2024. Demonstrating a Hyperledger Fabric-based Blockchain with Knowledge Graphs for a Supply Chain Ecosystem. *2024 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)* (2024), 15–16. doi:10.1109/ICBC59979.2024.10634370
[67] Aditya Sirish A Yelgundhalli, Patrick Zielinski, Reza Curtmola, and Justin Cappos. 2025. Rethinking Trust in Forge-Based Git Security. Network and Distributed System Security (NDSS) Symposium.

Received 28 August 2025; revised 11 January 2026; accepted 15 January 2026