The American Journal of Engineering and Technology
ISSN 2689-0984 | Open Access

TYPE Original Research
PAGE NO. 96-101
DOI 10.37547/tajet/Volume07Issue07-11

Check for updates

Inside Blockchain Startups: Precision Strategies to Sidestep Technical Pitfalls

Vladislav Markushin
Team Lead, Rust Developer at Composable Foundation, Argentina

OPEN ACCESS

SUBMITTED 07 June 2025
ACCEPTED 29 June 2025
PUBLISHED 24 July 2025
VOLUME Vol.07 Issue 07 2025

CITATION
Markushin, V. (2025). Inside Blockchain Startups: Precision Strategies to Sidestep Technical Pitfalls. The American Journal of Engineering and Technology, 7(07), 96–101. https://doi.org/10.37547/tajet/Volume07Issue07-11

Abstract: This paper presents a structured analysis of common engineering failures in early-stage blockchain startups. Drawing on practical experience within the Ethereum, Solana, Polkadot, and Cosmos SDK ecosystems, it identifies five recurrent categories of technical pitfalls: inadequate system design, fragile or overly coupled architectures, improper use of cross-chain protocols, insufficient build and release automation, and limited observability and runtime diagnostics. The study employs a case-based methodology across five representative projects, covering a wide range of protocol layers and architectural patterns—from decentralized messengers to cross-chain bridge infrastructure.

Findings demonstrate that the adoption of mature engineering practices—such as Command–Query Responsibility Segregation (CQRS), event sourcing, finite state machines, proxy contract standards (EIP-1967, EIP-2535), and type safety in Rust—substantially improves system resilience and extensibility. Particular emphasis is placed on Inter-Blockchain Communication (IBC) as a robust standard for secure interoperability across heterogeneous chains. The paper also highlights how automated CI/CD pipelines, multi-layer telemetry, and centralized alerting frameworks support early fault detection and operational responsiveness. The study concludes that minimizing custom low-level implementations in favor of standardized, modular approaches is critical for building secure, auditable, and scalable decentralized systems.

Keywords: blockchain, cross-chain, Ethereum, Solana, Polkadot, IBC, development

Introduction

Blockchain systems, characterized by decentralized

consensus and tamper-resistant data structures, impose a distinct set of design constraints on early-stage technology ventures. These constraints frequently surface in the form of technical missteps, such as premature optimization, custom protocol development, or the absence of robust architectural frameworks. Collectively, such missteps can delay system rollouts, introduce vulnerabilities, and erode a project's operational viability.

This study draws on my applied experience as a Rust engineer working within the Ethereum, Solana, Polkadot, and Cosmos SDK ecosystems. Projects analyzed span multiple blockchain domains, including messaging systems (Paymon Inc.), protocol-level consensus (Soramitsu), and cross-chain bridge architecture (Composable Foundation).

The analysis identifies five recurrent categories of technical pitfalls in blockchain startups: (1) Inadequate system design strategies; (2) Fragile or monolithic software architecture; (3) Misapplication or underutilization of cross-chain communication protocols; (4) Deficient build automation and release orchestration; (5) Limited observability and runtime introspection mechanisms. Each category is examined through a case study methodology grounded in real-world deployments, with the goal of offering practical mitigations informed by both general software engineering principles and blockchain-specific requirements. This work aims to inform practitioners and researchers invested in advancing the resilience and scalability of decentralized systems.

Methodology

This study adopts a qualitative, case-based research approach to examine recurring technical challenges in the development of blockchain infrastructure. The analysis is grounded in direct engineering experience across five representative projects, selected for their diversity in architectural scope, consensus mechanisms, and protocol layers.

- **Crypto Messenger (Paymon Inc., 2017–2019):** A messaging platform incorporating integrated cryptocurrency transfers. The project encountered issues related to architectural over-engineering and nonstandard protocol design.
- **Iroha Blockchain (Soramitsu, 2020–2021):** standalone Blockchain distributed ledger system. This case illustrates the challenges of managing protocol complexity and developing a secure smart contract environment.
- **Decentralized Loyalty System (Bitfury, 2019–2020):** A modular architecture implementing GOST cryptographic standards. The focus here is on system modularity and the implications of cryptographic design constraints on infrastructure extensibility.
- **Ethereum–Internet Computer Bridge (InfinitySwap, 2021–2022):** A cross-chain bridge protocol emphasizing system-level optimization and secure interoperability across heterogeneous chains.
- **IBC Bridges and Solana Rollup (Composable Foundation, 2022–present):** A multi-chain integration effort featuring Inter-Blockchain Communication (IBC) and rollup technologies. Particular attention is given to governance-aware architecture and formal state verification mechanisms.

Results and Discussion

System Design Pitfalls

A recurring design failure observed across early-stage blockchain projects involves the unnecessary reinvention of foundational infrastructure. In one case study, the development team elected to construct the entire network stack from first principles, bypassing established remote procedure call (RPC) protocols and serialization libraries. Instead, a custom binary protocol was developed in C++. While this approach was intended to offer greater control, it significantly impeded debugging, delayed feature delivery, and ultimately necessitated a complete architectural reimplementation.

This tendency — favoring bespoke solutions over standardized tooling — is prevalent among blockchain startups. In pursuit of performance optimization or architectural purity, teams frequently forgo widely adopted technologies such as JSON-RPC, WebSockets, or serialization libraries like borsh and scale-codec. In practice, however, these decisions often fragment development efforts, hinder interoperability, and result in the accumulation of long-term technical debt.

These findings reinforce a broader principle: in the initial

phases of system development, engineering priorities should be weighted toward observability, verifiability, and maintainability rather than micro-optimizations. Architectures based on robust, well-supported components are less prone to critical failure modes and reduce the operational overhead associated with long-term maintenance.

Architectural Practices and Change Resilience

Although architectural shortcomings may appear benign in the early stages of distributed system development, they often become major obstacles to scalability and evolvability as the system matures. Across the examined case studies, several architectural anti-patterns emerged repeatedly. These included reliance on mutable global state, insufficient modularization, tight coupling between functional components, and frequent violations of established object-oriented design principles—particularly those articulated by the SOLID framework. These deficiencies collectively increased the cognitive load required for refactoring, impeded the integration of new features, and elevated the risk of regressions and latent defects.

Several design strategies consistently proved effective in mitigating these issues:

- **CQRS (Command–Query Responsibility Segregation):** Partitioning read and write operations enabled clearer separation of concerns. In the context of cross-chain bridges, this approach allowed transaction submission and status queries to be processed through distinct layers, enhancing both performance and reliability.
- **Event Sourcing:** Persisting system state transitions as a stream of immutable events facilitated reliable state reconstruction and provided a natural integration point for downstream services through event subscription mechanisms.
- **Finite State Machines (FSMs):** Formalizing system behavior via FSMs provided a rigorous framework for managing operation lifecycles. This was particularly critical in cross-chain workflows, where strict sequencing constraints are required to prevent invalid or premature transitions.
- **Proxy Contract Patterns (EIP-1967) and the Diamond Standard (EIP-2535):** These upgradeability techniques enabled safe replacement of contract logic without jeopardizing the integrity of stored state—an essential requirement in financial systems handling on-chain assets.
- **Type safety in Rust:** The use of Rust's type system, including generics, traits, and algebraic data types, played a central role in enforcing invariants at compile time. This improved both modular reasoning and overall system reliability by catching potential logic errors prior to runtime.

The application of these patterns significantly improved maintainability, facilitated the addition of new features without architectural compromise, and enhanced testability. Collectively, these practices promote software systems that are resilient to change and robust against failure, which are key attributes in the rapidly evolving landscape of blockchain infrastructure.

Cross-Chain Protocols and IBC

Cross-chain integration presents stringent requirements in terms of data consistency, synchronization guarantees, and fault tolerance. Ensuring secure and reliable inter-network communication in a heterogeneous blockchain environment remains one of the most technically demanding aspects of decentralized systems engineering. Across multiple projects, implementation deficiencies in cross-chain bridge protocols frequently resulted in state divergence between participating networks, exposing users to loss of funds and undermining trust in protocol correctness.

The most reliable solution in this context was the Inter-Blockchain Communication (IBC) protocol. IBC formalizes inter-chain message passing through the Interchain Standards (ICS) specification suite, offering a rigorously defined interface for verifying remote states and coordinating cross-chain operations.

Key architectural modules include:

- **ICS-02: Light Clients**

  Responsible for verifying the state of remote blockchains through cryptographic validation of Merkle proofs and consensus headers. These clients play a central role in maintaining data integrity and detecting protocol violations such as equivocation

(double-signing) or timestamp manipulation.

- **ICS-03: Connection Handshake**

  Specifies a four-phase handshake protocol — ConnOpenInit, ConnOpenTry, ConnOpenAck, and ConnOpenConfirm — used to establish authenticated and validated communication channels between two chains. The handshake protocol relies on trusted client state for bidirectional agreement.

- **ICS-26: Middleware Routing**

  Provides a modular routing layer that enables protocol extensibility without modifying core application logic. This facilitates the seamless integration of optional features such as fee abstraction, on-chain routing policies, or interchain account management via custom middleware.

- **ICS-08: WASM Clients**

  Implements client-side verification logic using WebAssembly (WASM), allowing for more flexible upgrades and cross-language development. Smart contracts are deployed using MsgStoreCode and upgraded via MsgMigrateContract, enabling governance-based evolution of interchain clients without requiring network forks.

In practice, successful IBC deployment necessitates the development of chain-specific light client adapters to accommodate varying consensus mechanisms. Furthermore, strict typing of interchain state data and routine protocol audits are essential to ensuring correctness, resilience, and auditability.

Adherence to the IBC framework significantly reduces the likelihood of inconsistent state transitions or unexpected asset behavior, supporting the construction of secure and scalable cross-chain communication protocols.

Build Automation and Release Processes

Blockchain startups frequently operate under compressed development timelines, requiring rapid iteration, support for multiple deployment environments (such as development, testnet, and production), and strict reproducibility guarantees across heterogeneous platforms. In the absence of robust continuous integration and delivery (CI/CD) infrastructure, these operational pressures often result in unreliable builds, delayed release cycles, and elevated operational burden on engineering teams.

To address these challenges and enhance build reliability, the projects examined in this study adopted a fully automated approach to the software delivery lifecycle. Docker-based containerization was employed to ensure environmental consistency across development and production systems, with image publishing managed via the docker/build-push-action. This setup enabled reproducible builds and significantly accelerated the delivery of new releases. Build times were further optimized through the use of dependency caching in GitHub Actions; in particular, caching the Cargo and target directories with the rust-cache action proved highly effective for Rust-based systems with complex dependency graphs.

Automated enforcement of code quality standards was integrated into every CI stage. Static analysis was performed using cargo clippy, formatting compliance was verified via cargo fmt -- --check, and test coverage metrics were collected using cargo-llvm-cov. These tools ensured that potential issues were identified early in the development cycle—prior to code integration—thereby improving consistency and maintainability across the codebase.

Deployment workflows were managed through Amazon Web Services (AWS), with orchestration pipelines issuing commands such as aws ecs update-service and aws eks update-kubeconfig to perform seamless updates of containerized services and Kubernetes clusters. To prevent configuration drift across environments, deployment logic was isolated using github.ref conditions in conjunction with scoped environment variables.

Release processes were governed by semantic versioning, with Git tags in the v*.*.* format triggering automated deployment and rollback mechanisms. This workflow reduced the risk of human error, minimized manual intervention, and significantly tightened the feedback loop between development and production environments.

Monitoring and Observability Systems

In the early stages of one project, the absence of a structured observability framework significantly delayed the detection and resolution of critical production issues. Latency spikes in API responses, inconsistencies in transaction processing, and instances of CPU exhaustion were only recognized post hoc—typically following user reports or service disruptions. The lack of centralized monitoring and alerting mechanisms led to prolonged recovery times and diminished operational responsiveness.

To mitigate these challenges, observability practices were extended across the full system stack. At the infrastructure level, telemetry was collected for key resource indicators, including CPU utilization, memory allocation, disk input/output operations, and network throughput. These metrics were monitored both at the host and container levels to enable fine-grained resource tracking in containerized deployments.

At the application layer, operational telemetry encompassed API latency distributions, request throughput, error rates (HTTP 4xx/5xx), and transaction queue depth. These indicators were instrumental in assessing the health and responsiveness of core business logic components under varying load conditions.

In addition, blockchain-specific metrics were captured to provide insight into protocol-layer behavior. Monitored signals included current block height, reorganization frequency, mempool congestion, and gas consumption patterns. These metrics supported early detection of chain instabilities and facilitated preemptive remediation of network-level anomalies.

Centralized log aggregation was performed using the ELK stack and Grafana Loki, enabling efficient log retrieval and correlation during incident response. For deeper performance diagnostics, distributed tracing — implemented via OpenTelemetry and Jaeger — enabled end-to-end request flow analysis across microservices, facilitating root cause identification for latency and reliability regressions.

Complementing passive monitoring, synthetic blackbox checks were deployed to continuously verify the availability and responsiveness of critical endpoints, including RPC interfaces. Alerting workflows were managed through Alertmanager, with notification routing to operational communication platforms such as Slack and Telegram. Alerting logic was governed by service-level indicators (SLIs) and service-level objectives (SLOs), combining static thresholds with anomaly detection techniques to minimize false positives and prioritize events requiring immediate intervention.

Conclusion

Engineering failures in blockchain startups extend beyond project delays; they erode user trust, complicate maintainability, and in many cases, introduce vulnerabilities that compromise asset security. The root causes of such failures are varied, encompassing the unnecessary development of custom protocols, the absence of well-defined architectural patterns, and the lack of systematic observability and operational visibility.

The case studies presented in this paper highlight that resilience in blockchain infrastructure is not a product of novelty but of disciplined engineering. Systems that rely on mature, standardized protocols and libraries—rather than bespoke, low-level implementations—tend to exhibit greater maintainability and fault tolerance. Modular architectural designs with strict separation of concerns support adaptability and scalability, while formalized inter-chain communication via protocols such as IBC introduces robustness in cross-network interactions. Additionally, automation of build, deployment, and monitoring pipelines enhances reproducibility and operational responsiveness, minimizing downtime and human error.

Together, these practices offer a pragmatic foundation for the development of secure, auditable, and scalable blockchain systems. As the ecosystem continues to mature, adopting these engineering principles will be essential not only for technical excellence but also for maintaining long-term trust in decentralized technologies.

References

1. ConsenSys. (2021). *Poly Network Hack Analysis.* ConsenSys Diligence Report. Retrieved from https://consensys.net/diligence.

2. Cosmos. (2022). *Inter-Blockchain Communication Protocol Specification.* Cosmos SDK Documentation. Retrieved from https://docs.cosmos.network.

3. Dannen, C. (2017). *Introducing Ethereum and Solidity: Foundations of Cryptocurrency and Blockchain Programming for Beginners.* Apress.

4. Knuth, D. E. (1974). Structured Programming with go to Statements. *ACM Computing Surveys (CSUR)*, 6(4), 261–301.

5. Wood, G. (2016). *Polkadot: Vision for a Heterogeneous Multi-Chain Framework.* Web3 Foundation White Paper.