PREFACE TO THE EDITION

It is with great pleasure that we present the latest issue of the Eduschool Journal of Computer Science Research Studies (EJCSRS). This issue brings together a diverse collection of scholarly contributions that explore emerging paradigms, advanced architectures, and transformative technologies shaping the future of computer science. The selected articles reflect the rapid evolution of computing systems and the growing need for solutions that combine correctness, security, scalability, performance, and resilience.

The issue opens with an insightful examination of advanced type systems, exploring dependent types, linear types, and effect systems as mechanisms for achieving compile-time correctness guarantees. The article highlights how modern programming language research is extending the capabilities of type systems beyond error detection toward formal verification and functional correctness.

Reliability engineering in large-scale distributed environments is addressed through a comprehensive study of chaos engineering practices. By focusing on controlled fault-injection experiments, blast-radius management, and automated rollback mechanisms, the article demonstrates how organizations can proactively improve system resilience and prepare for unexpected operational failures.

The growing importance of analytical data processing is reflected in a detailed exploration of column-oriented database architectures. Through an analysis of ClickHouse and DuckDB, the study investigates vectorized execution, late materialization, and compression strategies that enable high-performance analytics on massive datasets.

Another significant contribution examines memory-safe systems programming and the migration of critical infrastructure from traditional C and C++ environments to Rust. The discussion offers valuable insights into ownership models, interoperability challenges, incremental migration strategies, and the broader implications of memory safety for modern software security.

Cloud-native computing receives focused attention through an in-depth comparative analysis of service mesh architectures in multi-cluster Kubernetes environments. By evaluating Istio and Linkerd, the article provides important perspectives on traffic management, security, observability, and operational trade-offs in contemporary microservices ecosystems.

Collectively, the articles in this issue demonstrate the breadth and dynamism of contemporary computer science research. They address foundational theoretical advances alongside practical engineering solutions, offering valuable knowledge for researchers, practitioners, educators, and students alike. As computing systems become increasingly interconnected, intelligent, and security-conscious, the themes explored in this issue provide important insights into the technologies that will shape the next generation of digital innovation.

We extend our sincere appreciation to all authors, reviewers, and editorial board members whose dedication and expertise have made this publication possible. We hope that the contributions presented herein stimulate further research, foster interdisciplinary dialogue, and inspire continued advancements in the field of computer science.

Dr. Sr. Mini T V
Chief editor

CONTENTS

EDUSCHOOL JOURNAL OF COMPUTER SCIENCE RESEARCH STUDIES (EJCSRS)

(Open Access, Double-Blind Peer Reviewed Journal)

ISSN Online:

Advanced Type Systems: Dependent Types, Linear Types, and Effect Systems for Compile-Time Correctness Guarantees

V Sakhtipriyanka

Assistant Professor, Department of Computer Science, Kongunadu Arts and Science College, Coimbatore, India

Article information

Received: 10th February 2026
Received in revised form: 18th March 2026
Accepted: 21st April 2026
Available online: 12th June 2026

Volume: 1
Issue: 2
DOI: https://doi.org/10.5281/zenodo.20654795

Abstract

A type checker is the most widely deployed program verifier in the world, run by millions of engineers who may never call it that. Most use only its weakest form, which catches little more than adding a number to a string. A richer tradition in programming-language research pushes the type system far further, until it can prove properties that today are checked, if at all, by tests and review. This paper surveys three of the most consequential of these advanced disciplines. Linear and affine types track the use of resources, guaranteeing that a file handle, a lock, or a block of memory is consumed exactly once, which eliminates leaks and use-after-free at compile time and underlies the ownership model that has recently reached mainstream systems languages. Effect systems make a function's side effects, its input and output, its exceptions, its access to state, part of its type, so the compiler can enforce that effects are handled and that pure code stays pure. Dependent types, the most powerful of the three, allow types to depend on values, collapsing the distinction between programs and proofs and making it possible to state and verify full functional correctness within the language itself. We explain the core idea behind each, trace how ideas once confined to proof assistants are entering production languages, and assess the cost, chiefly in type-checking complexity and programmer effort, that has kept the most powerful systems from widespread adoption.

Keywords:- Type Systems, Dependent Types, Linear Types, Affine Types, Effect Systems, Algebraic Effects, Formal Verification, Programming Languages, Curry-Howard Correspondence.

I. INTRODUCTION

Every statically typed language ships a proof checker, though few of its users think of it that way. When the compiler rejects a program because a function expecting an integer was handed a string, it has refuted a small theorem about that program's behaviour before the program ever runs [1]. The discipline of type theory asks an ambitious question: if a type checker can prove this, what else can it prove? The answer, developed over decades of research, is a great deal more than mainstream languages currently exploit, extending all the way to full functional correctness.

The appeal is the timing of the guarantee. A test demonstrates that a program behaved correctly on the inputs the test happened to try; a type proves a property for all inputs, and it does so at compile time, before deployment, with no runtime cost [1]. The catch is expressiveness: the simple type systems that are cheap to check can only express simple properties, and pushing toward richer guarantees means a more powerful, and more demanding, type system. Figure 1 sketches this spectrum, from untyped code through the polymorphism of ordinary functional languages to the advanced disciplines this paper concerns.

We examine three of those disciplines, chosen because each has moved, or is moving, from research into practice. Linear and affine types govern resource usage. Effect systems govern side effects. Dependent types govern logical correctness. Figure 2 summarises what each eliminates. The remainder of the paper treats them in turn, then weighs the adoption cost that determines how far each will spread.

II. LINEAR AND AFFINE TYPES: ACCOUNTING FOR RESOURCES

A. The Idea

Ordinary type systems treat values as freely copyable and discardable, which is exactly wrong for resources. A file handle should be closed once, not zero times and not twice; a lock acquired must be released; a buffer freed must not be touched again. Linear types, derived from Girard's linear logic, enforce that a value is used exactly once, and the affine variant relaxes this to at most once [6], [7]. Under such a system the compiler can guarantee, statically, that a resource is neither forgotten, which would be a leak, nor used after it has been consumed, which would be a use-after-free [6]. The accounting is done entirely at type-checking time and disappears at runtime.

B. From Theory to Mainstream Systems

For years this discipline lived mostly in research languages, but it has now reached production. Rust's ownership model is, in essence, an affine type system in practical clothing: a value has one owner, moving it invalidates the source, and the borrow checker enforces that references do not outlive what they point to, which is precisely how it eliminates the memory-safety errors that dominate systems-software incident reports [9]. Haskell has gained linear types as a first-class extension, allowing libraries to express must-use-once protocols in an otherwise non-linear language without splitting it into two [8]. The significance of these developments is that a once-esoteric corner of type theory turned out to be the key to safe, garbage-collector-free resource management at industrial scale [8], [9].

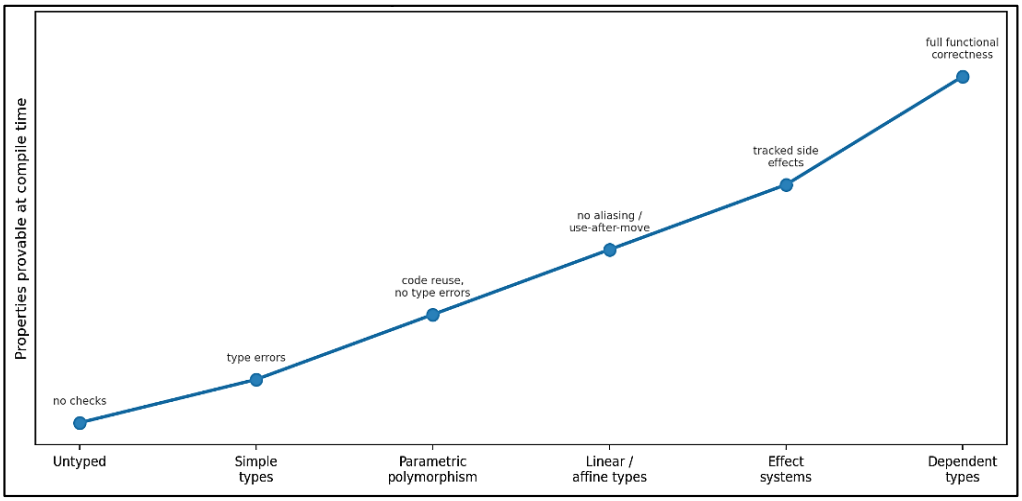

Fig 1: A spectrum of static guarantees. Each step rightward lets the type checker prove a stronger class of property, at the price of a more demanding type system.

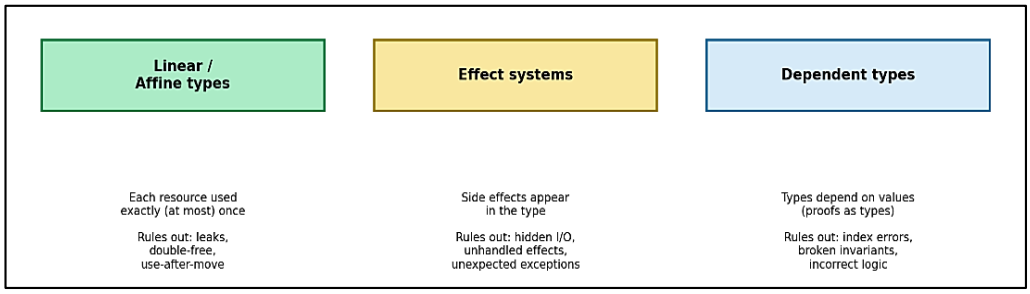

Fig 2: The three advanced disciplines compared by what each eliminates at compile time: linear and affine types account for resources, effect systems track side effects, and dependent types verify full functional correctness.

III. EFFECT SYSTEMS: MAKING SIDE EFFECTS VISIBLE

A. Types That Describe What a Function Does

A function's type ordinarily states what it takes and what it returns, but stays silent about what it does along the way, whether it reads a file, mutates global state, throws an exception, or blocks on the network. An effect system adds that missing information, annotating the type with the set of effects the function may perform [10]. Once effects are visible in types, the compiler can enforce useful disciplines: that a function declared pure performs no I/O, that every effect a program raises is eventually handled, and that effectful and pure code are not silently mixed. This turns a class of latent runtime surprises, an unexpected exception, an unhandled asynchronous operation, into compile-time errors [11].

B. Algebraic Effects and Handlers

The most influential modern formulation is algebraic effects with handlers, which separate the act of raising an effect from the code that interprets it, much as an exception separates a throw from its catch, but generalised to resumable operations [10], [11]. A program performs an abstract operation, and a surrounding handler decides what it means: in production it performs real I/O, in a test it returns canned responses, and the same code serves both because the effect and its interpretation are decoupled. This structure subsumes exceptions, mutable state, generators, and cooperative concurrency under one mechanism, generalising the earlier monadic treatment that first made side effects explicit in pure functional languages [15], and the type system tracks which operations a piece of code may perform [11], [12]. Languages designed around this idea, and extensions retrofitting it onto established ones, are an active and fast-moving area, because the same discipline that makes effects testable also makes concurrency and resource scoping easier to reason about [12].

IV. DEPENDENT TYPES: PROGRAMS AS PROOFS

A. Types That Depend on Values

The most powerful of the three disciplines erases the boundary between types and values. In a dependently typed language a type may mention a value: the type of vectors of length n, where n is an ordinary runtime number, or the type of sorted lists, or the type of a function that provably returns an index within bounds [2], [17]. Because types can now express arbitrary propositions about values, the type checker becomes a proof checker in the full sense, and a program that type-checks is a constructive proof that the stated property holds. This identification of propositions with types and proofs with programs is the Curry-Howard correspondence, and dependent type theory is its most complete realisation [2].

B. From Proof Assistants to Programming Languages

Dependent types first reached maturity in interactive theorem provers such as Coq and Agda, which are used to prove mathematical theorems and to verify software with machine-checked certainty [3], [4]. Two landmark results show what this enables: CompCert, a C compiler whose correctness is formally proved so that it provably preserves program behaviour, and seL4, an operating-system microkernel verified down to its implementation [13], [14]. A parallel effort has tried to make dependent types usable for ordinary programming rather than only proof, producing languages such as Idris that let an engineer state precise specifications and have the compiler enforce them as a normal part of writing code [5]. Recent work on quantitative type theory even unifies the resource tracking of linear types with the precision of dependent types in a single framework, suggesting the three disciplines of this paper are facets of one larger design space [16].

Table 1.Three Advanced Type Disciplines

Discipline	Core guarantee	Representative systems	Adoption stage
Linear / affine types	Resources used once; no leaks or use-after-move	Rust ownership, Linear Haskell	Mainstream (systems)
Effect systems	Side effects tracked and handled in types	Koka, OCaml effects, Eff	Emerging
Dependent types	Full functional correctness; proofs as programs	Coq, Agda, Idris, Lean	Research / verification

V. THE COST OF POWER

If these systems prove so much, the obvious question is why every language does not adopt them, and the answer is cost, paid in two currencies. The first is the difficulty of type checking itself. As types grow more expressive, deciding whether a program type-checks becomes harder, and in the fully dependent case type checking can require the programmer to supply proofs that the compiler cannot infer on its own [1], [5]. The

second currency is human effort. Writing a precise specification and convincing the type checker that the implementation meets it is real work, often comparable to writing the program, and it demands fluency in a style of reasoning that most engineers have never been taught [4], [5].

This cost explains the adoption pattern in Table 1. Linear and affine types reached the mainstream because they buy a large, concrete safety benefit, the elimination of memory errors, for a modest and well-contained increase in annotation burden, as Rust's success demonstrates [9]. Effect systems sit in between, valuable and increasingly practical but still settling into language designs [11], [12]. Full dependent types remain concentrated in verification, where the correctness of a compiler or a kernel justifies an extraordinary investment, rather than in everyday application code [13], [14]. The frontier of practical language design lies in lowering the second cost, finding ways to deliver more of the guarantee with less of the proof burden.

VI. CONVERGENCE

The three disciplines are often presented separately, but the more interesting story is their convergence. Rust took an idea from linear logic and made it ordinary. Mainstream functional languages are absorbing algebraic effects to tame concurrency and side effects. Theorem provers built on dependent types are verifying the very compilers and kernels that everything else runs on, and frameworks such as quantitative type theory show that resource tracking and dependent precision can live in one system [13], [14], [16]. The common thread is a steady migration of guarantees from runtime to compile time, and from external tools, tests, linters, separate verifiers, into the type system that engineers already run on every build. Each discipline is a different answer to the same question of what a compiler can be made to prove [1].

VII. REFINEMENT TYPES: A PRAGMATIC MIDDLE GROUND

Full dependent types are powerful but costly, and a body of work has sought most of the benefit at a fraction of the price through refinement types. A refinement type attaches a logical predicate to an ordinary type, describing not just that a value is an integer but that it is, say, a positive integer or an index strictly less than the length of a given array [19], [20]. The decisive engineering choice is to restrict these predicates to a logic that an automated solver can decide, so that an external satisfiability-modulo-theories solver discharges the proof obligations automatically rather than asking the programmer to write proofs by hand [20]. The result is a system that can verify the absence of out-of-bounds accesses, division by zero, and violated preconditions, properties firmly in the territory of dependent types, while keeping the programmer experience close to ordinary typed programming. Languages and extensions in this family, including refinement-typed dialects of mainstream functional languages and verification-oriented languages, have demonstrated the verification of real libraries with annotation rather than proof, which is why refinement types are among the most promising routes to bringing strong static guarantees into everyday code [19]. The cost reappears at the edges: predicates outside the solver's decidable fragment fall back on manual reasoning, and solver performance becomes part of the build's behaviour.

VIII. SESSION TYPES AND TYPED COMMUNICATION

The disciplines discussed so far govern a single program's values and effects, but distributed and concurrent systems fail most often at the seams, in the protocols by which components communicate. Session types extend the typing discipline to those protocols, describing the permitted sequence of messages on a communication channel as a type: send a request, then receive a response, then either repeat or close [21]. A program that type-checks against a session type is guaranteed to follow the protocol, which rules out whole categories of concurrency bug such as sending a message the other side does not expect or deadlocking on a mismatched exchange. The multiparty generalisation lifts this from two participants to many, deriving each participant's local protocol from a single global description and guaranteeing that, if every participant respects its local type, the ensemble cannot deadlock and communication stays well typed [22]. This is a striking demonstration that types can constrain not only what a program computes but how independent programs interact over time, and it connects the type-systems tradition directly to the correctness of distributed systems, where protocol mismatches are a leading source of failure [21], [22].

IX. TYPE INFERENCE AND THE USABILITY FRONTIER

Whether any of these disciplines reaches mainstream use depends as much on inference as on power, because a guarantee that demands heavy annotation will be resisted no matter how valuable it is. The Hindley-Milner system that underlies the ML and Haskell families occupies a famous sweet spot: it infers the most general type of an expression with no annotations at all, giving strong polymorphic typing for free [18]. The difficulty is that inference degrades as expressiveness grows. Adding subtyping, dependent types, or rich effects can make full inference undecidable, so the more powerful systems necessarily ask the programmer to supply more information [1], [18]. Bidirectional typing is the standard response, splitting the task into checking an expression against an expected type and synthesising a type where none is given, which recovers much of the convenience of inference

in settings where global inference is impossible and also yields more localised, comprehensible error messages [24]. A different compromise, gradual typing, lets static and dynamic typing coexist in one program so that a codebase can be migrated toward stronger guarantees incrementally rather than all at once [23]. Figure 3 places the disciplines of this paper on the two axes that govern adoption, expressive power against automation, and makes the central tension visible: the systems that prove the most tend to automate the least, and the research frontier is the effort to push points toward the upper right, delivering more guarantee for less burden. Table 2 records the correspondence that gives all of this its theoretical unity.

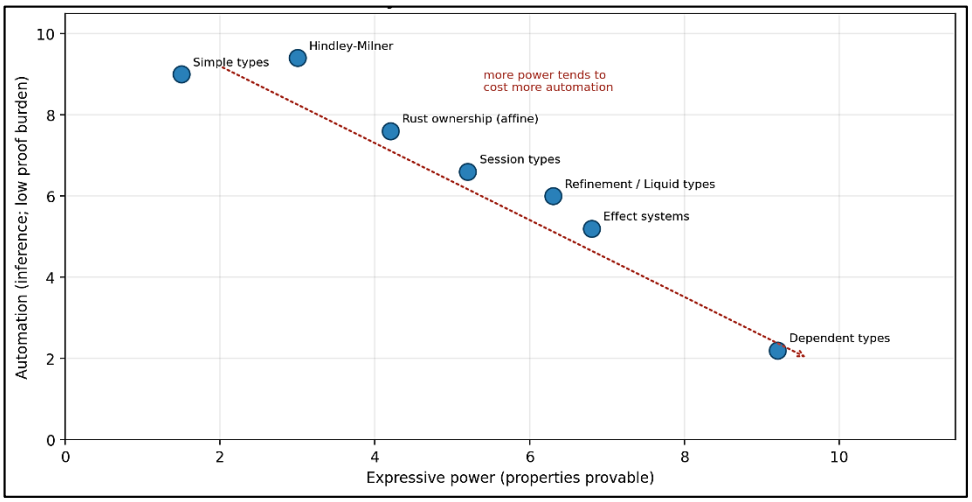

Fig 3: The usability frontier. Each discipline trades expressive power against the automation of type inference and the avoidance of manual proof; the practical research goal is to move systems toward the upper right.

Table 2. The Curry-Howard Correspondence

Logic	Type theory / programming	Consequence
Proposition	Type	A type states a claim about programs
Proof	Program (term)	Writing a program proves the claim
Implication A => B	Function type A -> B	A function transforms evidence
Conjunction / disjunction	Product / sum type	Pairing and choice of evidence
Universal quantifier	Dependent function type	Proofs parameterised by values

X. CONCLUSION

Advanced type systems represent the most practical bridge yet built between everyday programming and formal verification, because they deliver proofs through a mechanism every engineer already uses. Linear and affine types have crossed into the mainstream and now guarantee resource safety in production systems languages [8], [9]. Effect systems are crossing now, promising to make side effects and concurrency explicit and testable [11], [12]. Dependent types remain the demanding frontier, capable of proving full correctness but still costly enough to reserve for software whose failure is intolerable [5], [13], [14]. The trajectory is clear and consistent: properties once relegated to tests, documentation, and hope are moving, one discipline at a time, into the type checker, where the compiler enforces them on every build. The open engineering problem is not whether these guarantees are worth having but how to make them cheap enough that ordinary code can afford them.

REFERENCES

[1] B. C. Pierce, *Types and Programming Languages*. Cambridge, MA, USA: MIT Press, 2002.
[2] P. Martin-Löf, *Intuitionistic Type Theory*. Naples, Italy: Bibliopolis, 1984.
[3] Y. Bertot and P. Castéran, *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions*. Berlin, Germany: Springer, 2004.
[4] U. Norell, "Towards a practical programming language based on dependent type theory," Ph.D. dissertation, Dept. Comput. Sci. Eng., Chalmers Univ. Technol., Gothenburg, Sweden, 2007.
[5] E. Brady, "Idris, a general-purpose dependently typed programming language: Design and implementation," *J. Funct. Program.*, vol. 23, no. 5, pp. 552–593, 2013.
[6] P. Wadler, "Linear types can change the world!," in *Programming Concepts and Methods*, M. Broy and C. Jones, Eds. Amsterdam, The Netherlands: North-Holland, 1990, pp. 561–581.

[7] J.-Y. Girard, “Linear logic,” *Theoret. Comput. Sci.*, vol. 50, no. 1, pp. 1–101, 1987.
[8] J.-P. Bernardy, M. Boespflug, R. R. Newton, S. Peyton Jones, and A. Spiwack, “Linear Haskell: Practical linearity in a higher-order polymorphic language,” *Proc. ACM Program. Lang.*, vol. 2, no. POPL, pp. 1–29, 2018.
[9] N. D. Matsakis and F. S. Klock II, “The Rust language,” in *Proc. ACM SIGAda Annu. Conf. High Integrity Lang. Technol. (HILT)*, 2014, pp. 103–104.
[10] G. D. Plotkin and M. Pretnar, “Handlers of algebraic effects,” in *Proc. 18th Eur. Symp. Program. (ESOP)*, 2009, pp. 80–94.
[11] A. Bauer and M. Pretnar, “Programming with algebraic effects and handlers,” *J. Log. Algebr. Methods Program.*, vol. 84, no. 1, pp. 108–123, 2015.
[12] D. Hillerström and S. Lindley, “Liberating effects with rows and handlers,” in *Proc. 1st Int. Workshop Type-Driven Develop. (TyDe)*, 2016, pp. 15–27.
[13] X. Leroy, “Formal verification of a realistic compiler,” *Commun. ACM*, vol. 52, no. 7, pp. 107–115, 2009.
[14] G. Klein *et al*., “seL4: Formal verification of an OS kernel,” in *Proc. 22nd ACM Symp. Oper. Syst. Princ. (SOSP)*, 2009, pp. 207–220.
[15] P. Wadler, “Monads for functional programming,” in *Advanced Functional Programming*, Lecture Notes in Computer Science, vol. 925. Berlin, Germany: Springer, 1995, pp. 24–52.
[16] R. Atkey, “Syntax and semantics of quantitative type theory,” in *Proc. 33rd Annu. ACM/IEEE Symp. Logic Comput. Sci. (LICS)*, 2018, pp. 56–65.
[17] H. Xi and F. Pfenning, “Dependent types in practical programming,” in *Proc. 26th ACM SIGPLAN-SIGACT Symp. Princ. Program. Lang. (POPL)*, 1999, pp. 214–227.
[18] L. Damas and R. Milner, “Principal type-schemes for functional programs,” in *Proc. 9th ACM SIGPLAN-SIGACT Symp. Princ. Program. Lang. (POPL)*, 1982, pp. 207–212.
[19] N. Vazou, E. L. Seidel, R. Jhala, D. Vytiniotis, and S. Peyton Jones, “Refinement types for Haskell,” in *Proc. 19th ACM SIGPLAN Int. Conf. Funct. Program. (ICFP)*, 2014, pp. 269–282.
[20] P. M. Rondon, M. Kawaguchi, and R. Jhala, “Liquid types,” in *Proc. 29th ACM SIGPLAN Conf. Program. Lang. Design Implement. (PLDI)*, 2008, pp. 159–169.
[21] K. Honda, V. T. Vasconcelos, and M. Kubo, “Language primitives and type discipline for structured communication-based programming,” in *Proc. 7th Eur. Symp. Program. (ESOP)*, 1998, pp. 122–138.
[22] K. Honda, N. Yoshida, and M. Carbone, “Multiparty asynchronous session types,” in *Proc. 35th ACM SIGPLAN-SIGACT Symp. Princ. Program. Lang. (POPL)*, 2008, pp. 273–284.
[23] J. G. Siek and W. Taha, “Gradual typing for functional languages,” in *Proc. 7th Workshop Scheme Funct. Program.*, 2006, pp. 81–92.
[24] B. C. Pierce and D. N. Turner, “Local type inference,” *ACM Trans. Program. Lang. Syst.*, vol. 22, no. 1, pp. 1–44, 2000.

EDUSCHOOL JOURNAL OF COMPUTER SCIENCE RESEARCH STUDIES (EJCSRS)

(Open Access, Double-Blind Peer Reviewed Journal)

ISSN Online:

Chaos Engineering in Production: Designing Fault-Injection Experiments with Controlled Blast Radius and Automated Rollback

Navin Chandran

Senior Director, Cognizant Technology Solutions, USA

Article information

Received: 13th February 2026
Received in revised form: 20th March 2026
Accepted: 24th April 2026
Available online: 12th June 2026

Volume: 1
Issue: 2
DOI: https://doi.org/10.5281/zenodo.20655900

Abstract

Distributed systems fail in combinations no designer anticipated, and the failures that matter most are precisely the ones that integration tests never exercise: a dependency that slows rather than stops, a region that partitions, a retry storm that turns a small fault into an outage. Chaos engineering confronts this directly by injecting controlled failures into running systems, including production, to discover weaknesses before an uncontrolled failure does. The discipline is often misread as breaking things at random. It is the opposite: a scientific method applied to system resilience, in which an engineer states a hypothesis about steady-state behaviour, injects a specific fault, and measures whether the system holds up. This paper presents chaos engineering as an experimental practice and concentrates on the two mechanisms that make it safe enough to run on live traffic. Blast-radius control limits the fraction of users and infrastructure an experiment can affect, beginning with a single host and escalating only as confidence grows. Automated rollback continuously compares the live system against its expected behaviour and aborts the experiment the instant the deviation exceeds a threshold, bounding the worst case. We trace the practice from its origin in the Netflix Simian Army to today's platforms, describe how to design an experiment, and argue that deliberately injecting failure is now a core technique of reliability engineering rather than a stunt.

Keywords:- Chaos Engineering, Fault Injection, Resilience, Site Reliability Engineering, Blast Radius, Automated Rollback, Distributed Systems, Steady-State Hypothesis, Production Testing.

.

I. INTRODUCTION

A large distributed system is never fully healthy. At any moment some disk is failing, some network link is congested, some dependency is responding slowly, and the system is expected to absorb all of it without the user noticing [1]. The hard truth of operating such systems is that their resilience to these conditions is largely untested, because the conditions are difficult to reproduce in a test environment and emerge only from the interaction of components under real load [1], [4]. A system can pass every unit and integration test and still collapse the first time a single dependency degrades in production.

Chaos engineering was developed at Netflix to close this gap [1], [3]. Its premise is that the only reliable way to learn how a system behaves under failure is to make it fail, deliberately and under observation, while it is doing real work. The most famous early tool, Chaos Monkey, did exactly this by randomly terminating production instances during business hours, forcing engineers to build services that tolerate the loss of any single node as a routine event rather than a crisis [3], [15]. The practice has since matured from a single tool into a formal discipline

with stated principles, supported by commercial and open-source platforms and adopted well beyond its origin [1], [2].

The persistent misunderstanding is that chaos engineering means causing damage at random. It is better understood as the experimental method applied to operations, and its credibility rests on being safe to run where it matters most, in production [1]. This paper develops that view. We frame the practice as hypothesis-driven experimentation, then examine in detail the two safety mechanisms that make production experiments responsible, controlled blast radius and automated rollback, before discussing the platforms that implement them and the organisational conditions the practice requires.

II. CHAOS ENGINEERING AS EXPERIMENT

A. The Steady-State Hypothesis

A chaos experiment begins not with a fault but with a hypothesis, and this is what separates it from merely breaking things [1]. The engineer first defines the system's steady state in terms of a measurable output that indicates normal behaviour, such as the rate of successful requests, the latency distribution, or a business metric like streams started per second. The hypothesis is that this steady state will persist even when a specific fault is introduced. The experiment then injects that fault and tests the hypothesis against reality. If the steady state holds, the system has demonstrated resilience to that condition; if it does not, the experiment has found a weakness cheaply and on the team's terms rather than during an unplanned outage [1], [2]. Figure 1 shows this loop.

B. Realistic Faults, Real Environment

The faults injected are chosen to mirror events that actually occur in production: instances terminating, latency injected into a dependency, errors returned from a service, a network partition between zones, or resource exhaustion on a host [1], [14]. The principle that distinguishes mature chaos engineering is that these are best exercised in the production environment, because staging never faithfully reproduces production's scale, traffic patterns, and configuration, and a resilience result that holds only in staging proves little, which is why practitioners make the explicit case for testing resilience in production itself [1], [4], [5]. Running in production is also what makes the safety mechanisms indispensable, which is the subject of the next two sections. Research has pushed this further toward principled fault selection, using the system's own data-flow lineage to choose the failures most likely to expose a real weakness rather than testing combinations blindly [6], [7].

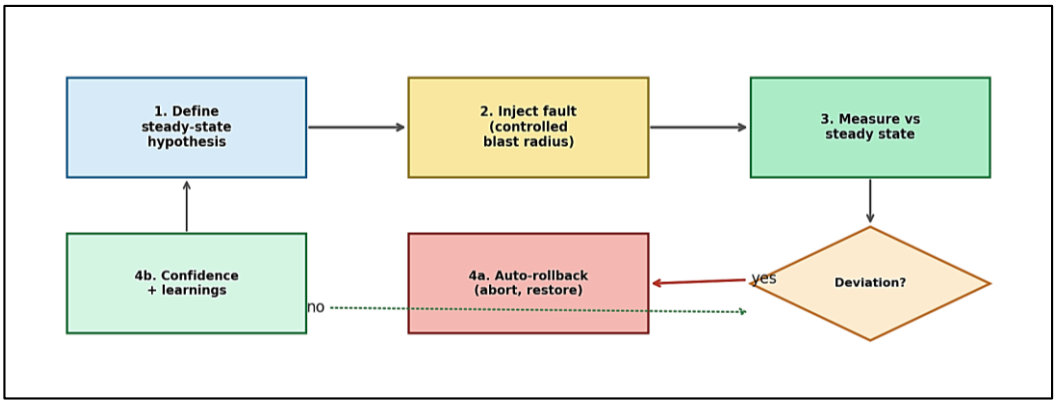

Fig 1: The chaos experiment loop. A steady-state hypothesis is tested by injecting a fault within a controlled blast radius; if measured behaviour deviates beyond threshold, the experiment is automatically rolled back, otherwise it yields confidence and learnings.

III. CONTROLLED BLAST RADIUS

A. Bounding Who Can Be Affected

The first safety mechanism is to limit the scope of any experiment, its blast radius, which is the set of users and infrastructure that a fault can possibly affect [2]. A well-designed experiment never begins at full scale. It starts with the smallest scope that can still produce a meaningful signal, a single host or a tiny fraction of traffic routed to a canary, and confirms the hypothesis there before any expansion [2], [10]. This containment is what makes it acceptable to experiment in production at all: even if the hypothesis is wrong and the system fails, the damage is confined to a deliberately small population, and the experiment has still surfaced the weakness.

B. Escalating With Confidence

Once an experiment passes at small scale, the blast radius is widened in stages, from one host to a single availability zone, to a region, and only eventually to the full system, with the hypothesis re-tested at each level

[2]. Figure 2 illustrates this escalation. The discipline is to increase exposure only as confidence accumulates, so that the experiments most likely to cause harm are run at the scale least able to do so. Targeting controls, routing the fault to specific users, devices, or request classes, let an experiment hold its blast radius precisely while still exercising a realistic slice of production traffic [2], [11]. The combination of small starting scope and gradual, evidence-gated escalation is what turns production fault injection from recklessness into engineering.

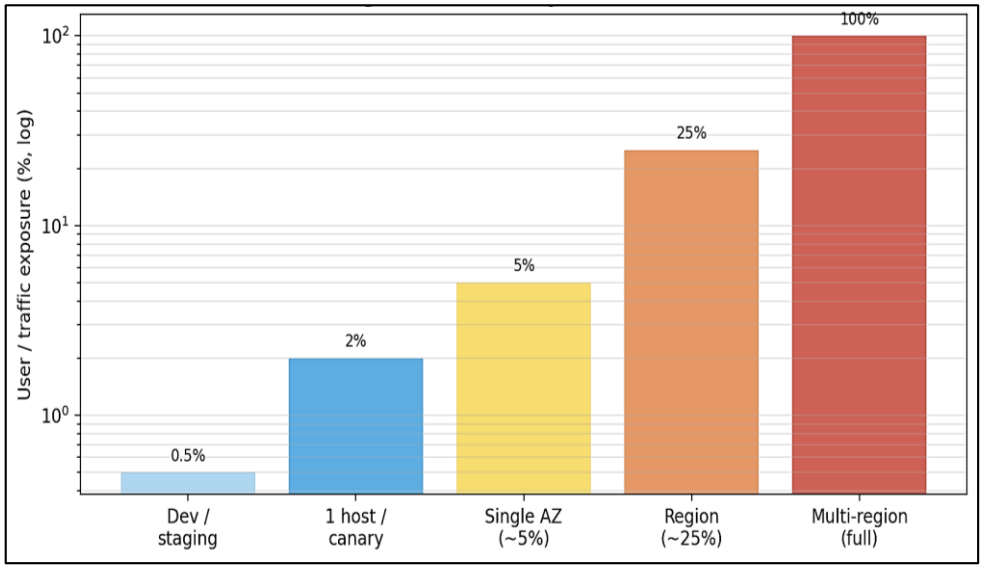

Fig 2: Blast radius is escalated only as each stage confirms the steady-state hypothesis, so the riskiest experiments run at the smallest scale and exposure grows with confidence.

IV. AUTOMATED ROLLBACK

Limiting blast radius bounds how many users an experiment can reach; automated rollback bounds how long and how badly it can affect them. The mechanism continuously monitors the steady-state metrics during the experiment and compares the population exposed to the fault against an unaffected control [1], [2]. The moment the deviation crosses a predefined threshold, the harm to the experimental group exceeding what the team has decided is acceptable, the system automatically aborts: it stops injecting the fault and restores normal conditions, without waiting for a human to notice and react [2], [14]. This automatic kill switch is essential because the value of chaos engineering depends on running real faults against real users, and no responsible team would accept that risk if the worst case were unbounded. With automated rollback the worst case is bounded by the detection threshold and the speed of the abort, both of which are engineered in advance rather than left to on-call reflexes. Together with blast-radius control, it transforms the potential cost of an experiment from open-ended to small and known, which is the precondition for experimenting in production responsibly [1], [2].

Table 1. The Two Safety Mechanisms of Production Chaos Experiments

Mechanism	Bounds	How	Failure if absent
Controlled blast radius	How many can be affected	Start tiny; escalate on confidence	A wrong hypothesis hits everyone
Automated rollback	How long / how badly	Threshold on steady-state deviation	A bad experiment runs until noticed
Steady-state hypothesis	What counts as failure	Measurable normal-behaviour metric	No criterion to abort against
Control group	Attribution of impact	Compare exposed vs unexposed	Cannot tell fault from noise

V. PLATFORMS AND TOOLING

The practice has been productised. Netflix generalised Chaos Monkey into a broader Simian Army and later into platforms that automate experiment design and execution at scale, including the data-driven selection of which failures to test [3], [7]. Commercial offerings package fault injection, blast-radius controls, and automated halting behind a managed interface, lowering the barrier for teams without Netflix-scale engineering [11]. In the cloud-native ecosystem, open-source projects integrate chaos experiments directly into container orchestration, expressing faults and their scope declaratively alongside the rest of a system's configuration [12]. What these tools share is an implementation of the same two safety mechanisms: every credible platform provides a way to bound

the blast radius and a way to halt automatically when steady-state metrics degrade [2], [11], [12]. The tooling, in other words, encodes the discipline rather than replacing it.

VI. THE ORGANISATIONAL DIMENSION

Chaos engineering is as much a cultural practice as a technical one, and it fails without organisational support. Deliberately injecting failure into production requires a blameless culture in which the weaknesses an experiment uncovers are treated as learning rather than fault, and it requires investment in the observability that makes steady state measurable and deviation detectable in the first place [4], [10]. The underlying philosophy is that of antifragility, the idea that a system subjected to controlled stress becomes stronger because it is continually forced to remove its weaknesses, rather than accumulating hidden fragility until a large uncontrolled failure exposes it all at once [4], [13]. Reliability engineering frameworks treat this proactive failure testing as a complement to error budgets and service-level objectives, a way to spend a controlled amount of reliability to buy confidence [10]. The practice connects, finally, to the longer tradition of dependable-systems research, which has always argued that fault tolerance must be validated by fault injection rather than assumed from design [8], [9], [16].

VII. A TAXONOMY OF FAULTS TO INJECT

Designing experiments requires a vocabulary of failures, and mature practice organises the faults worth injecting into a small number of domains, each exercising a different resilience property. Infrastructure faults remove compute: terminating an instance, draining a node, or simulating the loss of an entire availability zone, which tests whether the system tolerates the disappearance of capacity it was assumed to have [3], [14]. Network faults are often the most revealing, because the network is far less reliable than designers assume; injecting latency, packet loss, or a partition between zones probes the timeout, retry, and failover logic that only ever runs under exactly these conditions [17]. Resource faults exhaust CPU, memory, disk, or I/O on a host to test graceful degradation under pressure rather than outright failure. Application faults make a service return errors or respond slowly, and dependency faults degrade a downstream service, which together test the calling system's circuit breakers, fallbacks, and bulkheads, the very mechanisms meant to stop one component's trouble from becoming everyone's [9]. Figure 3 arranges these domains, and Table 2 pairs each with the resilience it is designed to validate. The discipline of the taxonomy is to ensure coverage: a resilience claim is only as trustworthy as the breadth of fault domains it has actually survived, and a program that only ever kills instances has tested just one corner of the space [6], [7].

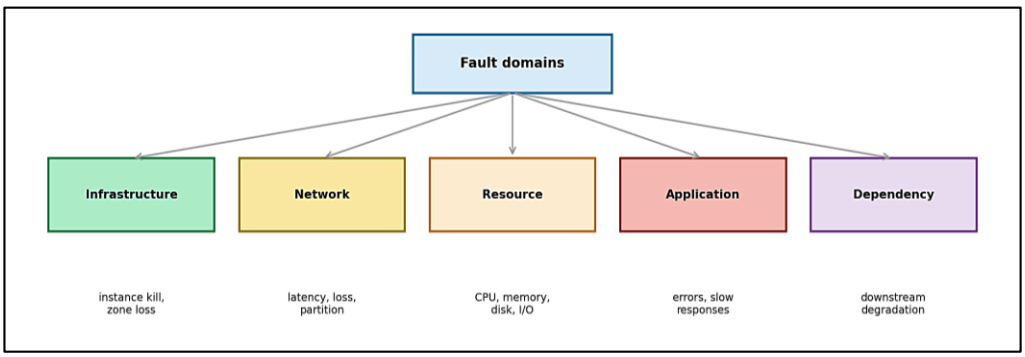

Fig 3: Faults worth injecting fall into a few domains, each validating a distinct resilience property. Broad coverage across domains is what makes a resilience claim credible; testing only one domain proves only one thing.

VIII. FROM GAMEDAYS TO CONTINUOUS CHAOS

Chaos engineering is usually adopted in stages, and understanding the progression helps a team place itself and plan its next step. The entry point is the GameDay, a scheduled, supervised exercise in which engineers gather to run a planned experiment against a system, watch how it and they respond, and capture the learnings, including gaps in tooling, runbooks, and alerting that the exercise exposes [2], [5]. GameDays are valuable precisely because they rehearse the human response alongside the technical one, but they are episodic, and a system changes continuously between them. The next stage automates individual experiments so they can run on demand with the safety mechanisms enforced programmatically, and the mature stage integrates them into the delivery pipeline as continuous verification, so that resilience is re-tested automatically as the system evolves rather than validated once and assumed to persist [2], [18]. Expressing experiments declaratively, as chaos as code checked into version control alongside the system, makes them repeatable, reviewable, and safe to run unattended within their configured blast radius [12], [19]. The trajectory mirrors the broader shift in operations from manual, periodic

checks toward automated, continuous assurance, and it is what lets resilience keep pace with the rate of change in a system that is never finished [10].

IX. MEASURING RESILIENCE

An experiment is only as good as the measurement that decides its outcome, and this is where chaos engineering depends utterly on observability. The steady-state hypothesis must be expressed as a metric the system actually emits, and detecting deviation in time to roll back requires that metric to be available with low latency and compared against an unaffected control to separate the fault's effect from normal variation [1], [2]. Beyond the per-experiment verdict, organisations track aggregate resilience indicators such as the time to detect and the time to recover from injected failures, using improvements in these as evidence that the practice is making the system more robust over time [10]. The connection to reliability engineering is direct: error budgets quantify how much unreliability is acceptable, and a chaos experiment is a controlled way to spend a small, deliberate portion of that budget to buy confidence, rather than discovering the system's limits through an uncontrolled outage that spends the budget all at once [10]. Care is required in the measurement itself, since tail latency and coordinated effects can mislead a naive metric, and the resilience-engineering tradition cautions that resilience is a property of the whole socio-technical system, the people and processes as well as the software, so the human responses an experiment reveals are as much a part of the result as the graphs [16], [20]. Academic work on systematic resilience testing reinforces the same point, showing that principled, repeatable fault injection finds weaknesses that ad hoc testing misses [21].

Table 2. Fault Domains and the Resilience Each Validates

Fault domain	Example fault	Resilience it tests
Infrastructure	Instance / zone termination	Tolerance of lost capacity, failover
Network	Latency, loss, partition	Timeouts, retries, partition handling
Resource	CPU / memory / disk exhaustion	Graceful degradation under pressure
Application	Injected errors or slow responses	Error handling, fallbacks
Dependency	Downstream service degradation	Circuit breakers, bulkheads

X. CONCLUSION

Chaos engineering reframes failure from something to be feared into something to be scheduled. By stating a steady-state hypothesis, injecting a realistic fault, and measuring the result, it applies the experimental method to the resilience of systems that are otherwise tested only by the accidents they suffer [1]. Its legitimacy as a production practice rests entirely on two mechanisms working together: a controlled blast radius that bounds how many users an experiment can affect and escalates only with confidence, and automated rollback that bounds how long and how severely by aborting the instant behaviour deviates [2]. With those guardrails, the worst case of an experiment is small and known in advance, which is what makes deliberately breaking a live system a responsible engineering act. From a single monkey terminating instances to declarative, data-driven platforms, the trajectory has been toward making controlled failure a routine part of building systems that must not fail [3], [7], [12]. The systems that survive their worst days are, increasingly, the ones that have already rehearsed them.

REFERENCES

[1] A. Basiri, N. Behnam, R. de Rooij, L. Hochstein, L. Kosewski, J. Reynolds, and C. Rosenthal, “Chaos engineering,” IEEE Software, vol. 33, no. 3, pp. 35–41, May–Jun. 2016.

[2] C. Rosenthal and N. Jones, Chaos Engineering: System Resiliency in Practice. Sebastopol, CA, USA: O’Reilly Media, 2020.

[3] Y. Izrailevsky and A. Tseitlin, “The Netflix Simian Army,” Netflix Technology Blog, Jul. 2011.

[4] [4] A. Tseitlin, “The antifragile organization,” Communications of the ACM, vol. 56, no. 8, pp. 40–44, Aug. 2013.

[5] J. Allspaw, “Fault injection in production: Making the case for resilience testing,” ACM Queue, vol. 10, no. 8, pp. 30–35, Aug. 2012.

[6] P. Alvaro, J. Rosen, and J. M. Hellerstein, “Lineage-driven fault injection,” in Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, VIC, Australia, 2015, pp. 331–346.

[7] P. Alvaro et al., “Automating failure testing research at internet scale,” in Proceedings of the 7th ACM Symposium on Cloud Computing (SoCC), Santa Clara, CA, USA, 2016, pp. 17–28.

[8] H. S. Gunawi et al., “FATE and DESTINI: A framework for cloud recovery testing,” in Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI), Boston, MA, USA, 2011, pp. 238–252.

[9] P. Joshi, H. S. Gunawi, and K. Sen, “PREFAIL: A programmable tool for multiple-failure injection,” in Proceedings of the ACM International Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), Portland, OR, USA, 2011, pp. 171–188.

[10] B. Beyer, C. Jones, J. Petoff, and N. R. Murphy, Eds., Site Reliability Engineering: How Google Runs Production Systems. Sebastopol, CA, USA: O'Reilly Media, 2016.
[11] Gremlin, Inc., "Gremlin: Enterprise chaos engineering platform," Gremlin Product Documentation, 2023.
[12] Cloud Native Computing Foundation, "Chaos Mesh: A cloud-native chaos engineering platform," CNCF Project Documentation, 2023.
[13] N. N. Taleb, Antifragile: Things That Gain from Disorder. New York, NY, USA: Random House, 2012.
[14] C. Bennett and A. Tseitlin, "Chaos Monkey released into the wild," Netflix Technology Blog, Jul. 2012.
[15] L. Hochstein, "The discipline of chaos engineering," in Seeking SRE, D. Blank-Edelman, Ed. Sebastopol, CA, USA: O'Reilly Media, 2018.
[16] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," IEEE Transactions on Dependable and Secure Computing, vol. 1, no. 1, pp. 11–33, Jan.–Mar. 2004.
[17] P. Bailis and K. Kingsbury, "The network is reliable: An informal survey of real-world communications failures," ACM Queue, vol. 12, no. 7, pp. 48–58, Jul. 2014.
[18] Chaos Community, "Principles of chaos engineering," principlesofchaos.org, 2018.
[19] R. Miles, Learning Chaos Engineering: Discovering and Overcoming System Weaknesses Through Experimentation. Sebastopol, CA, USA: O'Reilly Media, 2019.
[20] D. D. Woods, "Four concepts for resilience and the implications for the future of resilience engineering," Reliability Engineering & System Safety, vol. 141, pp. 5–9, Sep. 2015.
[21] V. Heorhiadi, S. Rajagopalan, H. Jamjoom, M. K. Reiter, and V. Sekar, "Gremlin: Systematic resilience testing of microservices," in Proceedings of the IEEE 36th International Conference on Distributed Computing Systems (ICDCS), Nara, Japan, 2016, pp. 57–66.

EDUSCHOOL JOURNAL OF COMPUTER SCIENCE RESEARCH STUDIES (EJCSRS)

(Open Access, Double-Blind Peer Reviewed Journal)

ISSN Online:

Column-Oriented Database Internals: Vectorized Execution, Late Materialization, and Compression in Clickhouse and Duckdb

Win Mathew John

Associate professor, PG Department of Computer Applications, Marian College Kuttikkanam (Autonomous), Kerala, India

Article information

Received: 17th February 2026
Received in revised form: 23rd March 2026
Accepted: 25th April 2026
Available online: 12th June 2026

Volume: 1
Issue: 2
DOI: https://doi.org/10.5281/zenodo.20657522

Abstract

The database systems that power analytics look almost nothing inside like the ones that power transactions, even when they speak the same SQL. The difference begins with a single decision about how a table is laid out on storage. A transactional system stores each row contiguously, which suits writing and reading whole records; an analytical system stores each column contiguously, which suits scanning a few columns across millions of rows. That one choice cascades into a distinct set of internal techniques that together make modern analytical engines an order of magnitude faster than row stores on the queries they target. This paper examines three of those techniques as realised in two influential open-source systems, ClickHouse and DuckDB. Vectorized execution processes data in batches of thousands of values rather than one tuple at a time, amortising interpreter overhead and unlocking the data-parallel instructions of modern CPUs. Late materialization keeps data in compact column form for as long as possible, delaying the expensive reconstruction of rows until it is unavoidable. And because a column holds values of one type with much local redundancy, lightweight compression schemes shrink storage and, critically, let the engine operate directly on compressed data. We explain the mechanism behind each, show how they reinforce one another, and discuss why the column store has become the default architecture for analytical workloads.

Keywords:- Column-Oriented Database, OLAP, Vectorized Execution, Late Materialization, Lightweight Compression, ClickHouse, DuckDB, Query Processing, Analytical Databases.

I. INTRODUCTION

Two kinds of work pull a database in opposite directions. Transaction processing inserts, updates, and retrieves individual records, touching every column of a few rows. Analytics scans enormous numbers of rows but only a handful of columns, computing aggregates and grouping results [3]. A storage layout optimal for one is poor for the other, and for decades the same row-oriented engines tried to serve both, to the detriment of analytics.

The column store resolves the tension by inverting the layout. Instead of storing each row's fields together, it stores each column's values together, so that scanning one attribute across the whole table reads a single contiguous run and skips the columns a query does not mention entirely [1], [2]. Figure 1 contrasts the two layouts. This idea, proven at research scale by C-Store and at production scale by a generation of systems that followed, is now the consensus architecture for analytical databases [2], [3]. But the layout is only the beginning. Its real power comes from the execution techniques it enables, because once data is stored by column, the engine can process it in ways a row store cannot.

This paper focuses on three such techniques, using ClickHouse and DuckDB as concrete reference points. ClickHouse is a distributed column store built for high-throughput analytics over very large datasets; DuckDB is an embeddable analytical engine that brings the same internals into a single process for interactive analysis [8], [9]. Despite their different deployment models, both rest on vectorized execution, late materialization, and column compression. We treat each in turn and show how, combined, they account for the performance gap illustrated in Figure 2.

II. THE STORAGE LAYOUT AND WHY IT MATTERS

The case for column storage on analytical workloads rests on the memory hierarchy. A query that averages one column over a billion rows reads, in a column store, exactly that column's data and nothing else, so no bandwidth is wasted moving fields the query ignores [1], [3]. In a row store the same query drags every column of every row through the cache to reach the one it needs. Because analytical queries are typically bandwidth-bound, eliminating that waste is a large and direct win. The layout also concentrates values of a single type and similar magnitude next to one another, which, as Section V explains, is what makes aggressive compression possible [7]. The foundational comparison of column and row stores quantified these effects and established that the advantage comes not from one trick but from the layout enabling a whole family of them [1]. The same columnar principle underlies the open storage formats and in-memory standards of the modern analytical ecosystem, from the nested columnar representation pioneered by Dremel [13] to the cross-language Apache Arrow layout that lets engines share columnar data without serialisation [14].

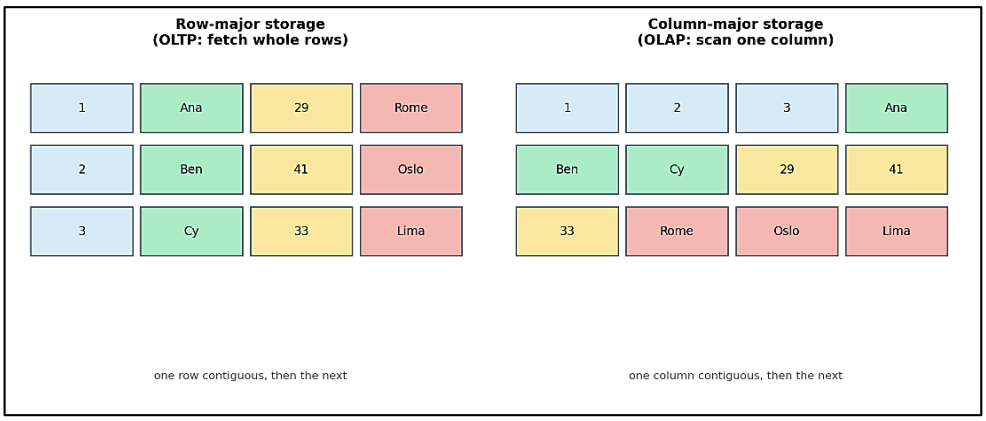

Fig 1: Row-major versus column-major storage of the same table. Scanning one attribute across all rows reads a contiguous run in the column layout and touches no irrelevant data.

III. VECTORIZED EXECUTION

A. Past the Tuple-at-a-Time Bottleneck

Classical query engines use an iterator model in which each operator produces one tuple at a time, pulling from its child on demand [16]. The model is elegant and composable, but at the scale of analytics it is ruinously slow, because every operator invocation carries a fixed overhead of function calls and interpretation that is paid once per row across billions of rows [4], [11]. Vectorized execution, pioneered in the MonetDB/X100 system, replaces the single tuple with a vector, a batch of typically a few thousand values from one column, as the unit that flows between operators [4], [5]. The per-batch overhead is now amortised over thousands of values, and the inner loop that does the real work becomes a tight pass over a contiguous array.

B. Friendly to the Hardware

Tight loops over typed arrays are exactly what modern CPUs reward. They keep the instruction pipeline full, make memory access predictable so the prefetcher succeeds, and let the compiler emit single-instruction multiple-data operations that apply one instruction to many values at once [5], [12]. This is why a column store's columnar layout and vectorized execution are natural partners: the data is already a contiguous typed array, which is the ideal input for a vectorized, SIMD-friendly kernel [3], [12]. An alternative school compiles each query to native code rather than interpreting vectors, and a careful study of the two approaches found they reach broadly comparable performance by different routes, with vectorization favouring simplicity and compilation favouring the very hottest paths [10], [11]. Both ClickHouse and DuckDB are built around vectorized execution [8], [9].

IV. LATE MATERIALIZATION

A query eventually has to produce rows, but the longer an engine can postpone assembling them, the faster it runs, and this is the insight behind late materialization [6]. In the naive, early-materialization strategy, the engine reconstructs full tuples from the relevant columns at the start of execution and then processes rows; in the late strategy, it carries the columns separately and operates on them in column form, stitching values into rows only at the latest possible moment, often after filtering and aggregation have already discarded most of the data [6]. The payoff is twofold. The engine does its work on compact, type-uniform column data that vectorizes well, and it never pays to reconstruct rows that a predicate later eliminates. Position lists and selection vectors track which rows survive each operator, so a filter can mark qualifying positions without materialising anything, and only the columns actually needed in the result are ever assembled [3], [6]. Late materialization is therefore not an isolated trick but the discipline that lets vectorized execution and compression keep paying off deep into a query plan.

V. COMPRESSION THAT EXECUTION CAN SEE THROUGH

A. Lightweight, Type-Aware Schemes

Because a column holds values of one type, often sorted or slowly changing, it compresses far better than a mixed row, and analytical engines exploit this with lightweight schemes chosen per column [7]. Run-length encoding collapses long stretches of a repeated value; dictionary encoding replaces a small set of distinct strings with compact integer codes; delta and frame-of-reference encodings store small differences instead of full values for sorted or clustered numeric columns; bit-packing uses only as many bits as a column's range requires [3], [7]. These are deliberately cheaper than general-purpose compressors, because the goal is not maximal ratio but the best balance of size against decompression speed.

B. Operating on Compressed Data

The decisive advantage is that some of these encodings can be processed without being decoded. A count or a filter over a run-length-encoded column can work on the runs directly; an equality predicate over a dictionary-encoded column can compare integer codes instead of strings; a sum can sometimes be computed from compressed representations [7]. This means compression reduces not only storage and the bytes read from disk but also the work the CPU performs, turning what is usually a space-time trade-off into a win on both axes [7], [12]. The integration of compression with execution, rather than treating it as a separate layer below the engine, is one of the defining internal characteristics of a modern column store, and both ClickHouse and DuckDB implement a palette of such encodings selected to keep vectorized kernels fed [8], [9].

Table 1. Lightweight Column Encodings and Their Properties

Encoding	Best for	Operate on compressed?	Typical use
Run-length (RLE)	Long repeated runs	Yes (count, filter)	Sorted / low-cardinality columns
Dictionary	Few distinct values	Yes (compare codes)	Strings, categoricals
Delta / frame-of-reference	Sorted or clustered numerics	Partially	Timestamps, sequential keys
Bit-packing	Small value ranges	Partially	Bounded integers, codes

VI. HOW THE PIECES COMPOUND

The three techniques are not independent optimisations stacked on a shared base; they reinforce one another, which is why their combined effect exceeds the sum of the parts shown in Figure 2. The columnar layout makes data a contiguous typed array, which is what vectorized execution needs and what compresses well [1], [3]. Compression keeps that array small enough to stay in cache and lets the engine compute over encoded values, feeding the vectorized kernels with less data and less work [7], [12]. Late materialization ensures the engine stays in this efficient column-and-vector regime for as long as possible, reconstructing rows only after most data has been filtered away [6]. Remove any one and the others lose force: vectorization over decompressed early-materialized rows would surrender much of the benefit. The architecture is best understood as a single coherent design in which storage layout, execution model, materialization strategy, and compression were chosen together, a lesson reinforced by commercial engines whose column-store acceleration integrates the same techniques [3], [15].

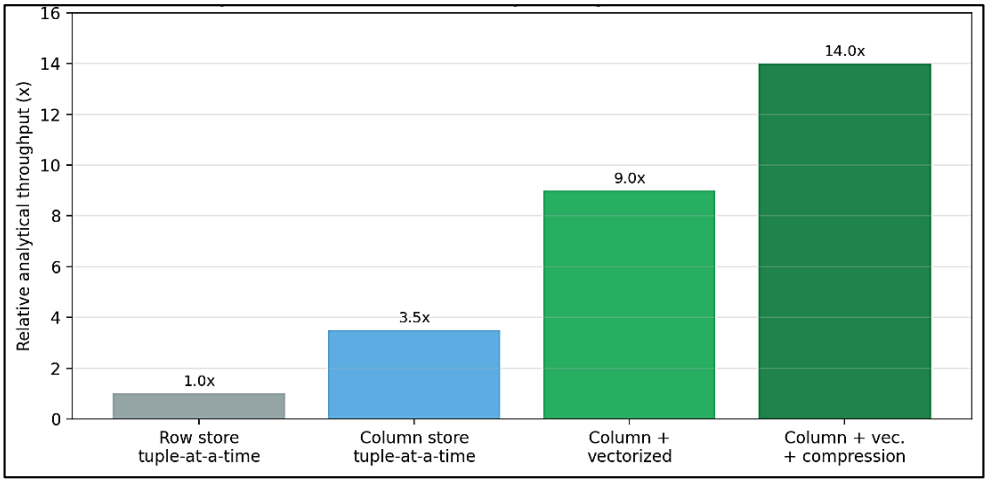

Fig 2: Representative analytical throughput as the column-store techniques are added in turn. The gains compound because the layout, vectorization, and compression reinforce one another rather than acting independently.

VII. INDEXING, ZONE MAPS, AND DATA SKIPPING

The fastest data to process is the data never read, and column stores invest heavily in avoiding irrelevant input. Rather than the fine-grained secondary indexes of transactional systems, which would be ill-suited to scan-heavy analytics, they rely on lightweight metadata that lets the engine skip large blocks of data whose values cannot match a query. The basic device is the zone map, a small summary, typically the minimum and maximum value, kept for each block of a column; when a query filters on that column, the engine consults the zone maps and reads only the blocks whose value range overlaps the predicate, skipping the rest without touching them [19]. Figure 3 illustrates the mechanism. This is especially effective on columns that are sorted or naturally clustered, such as timestamps in an append-mostly table, where a time-range query can eliminate the overwhelming majority of blocks. ClickHouse builds on this idea with a sparse primary index that marks the boundaries of granules along the table's sorting key, and with additional data-skipping indexes including bloom filters for membership tests on high-cardinality columns [9], [20], [22]. DuckDB maintains comparable zone-map statistics to prune row groups during a scan [8]. The common principle is that a little precomputed metadata converts a full scan into a partial one, compounding with the vectorized, compressed execution of the earlier sections.

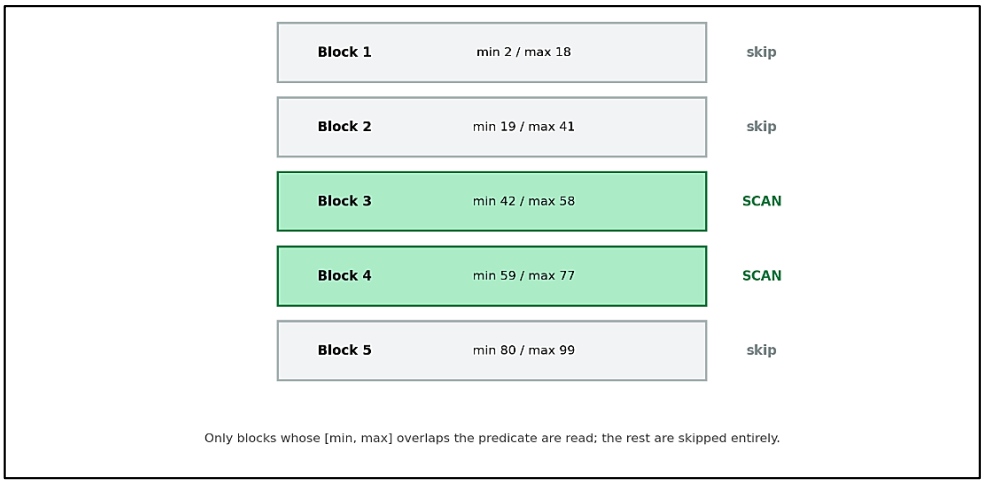

Fig 3: Zone maps turn a full scan into a partial one. Each block carries its value range, and only blocks whose range overlaps the query predicate are read, which is highly effective on sorted or clustered columns.

VIII. MERGE-BASED STORAGE AND HANDLING WRITES

Column stores optimise for reading, which raises the question of how they handle writes, since inserting a single row would otherwise require touching every column file. The dominant answer borrows the log-structured merge discipline from write-optimised key-value stores [18]. Incoming data is written in self-contained sorted parts rather than updated in place, and a background process periodically merges smaller parts into larger ones, keeping data ordered by the sorting key and the per-block metadata accurate [18], [22]. ClickHouse names its principal engine family for exactly this merge behaviour, and the choice of sorting key is the single most

consequential physical-design decision, because it determines both compression effectiveness and the power of data skipping [9], [22]. Updates and deletes, awkward in an append-oriented design, are handled by writing markers or replacement parts that the merge process later reconciles, so the system favours high-throughput appends and bulk loads over frequent small mutations [22]. This is a deliberate trade: analytical engines accept weaker support for in-place updates in exchange for the read performance that sorted, compressed, immutable parts make possible, which is appropriate for the append-mostly nature of analytical data [21].

IX. PARALLELISM, DISTRIBUTION, AND HARDWARE

The techniques described so far make a single core fast; modern engines must also exploit many cores and, for the largest data, many machines. Within a node, analytical engines parallelise a query by partitioning its input into chunks processed concurrently across cores, with a scheduler balancing the work so that no core sits idle, an approach exemplified by morsel-driven execution that has become the template for in-memory analytical parallelism [17]. The columnar, vectorized design is well matched to this, because vectors of values are natural units to distribute and recombine. DuckDB applies exactly this model to extract full performance from a single multi-core machine, which is often sufficient given how much modern hardware a laptop or server now provides [8], [17]. For data beyond one machine, ClickHouse shards tables across nodes and executes queries in a distributed fashion, pushing filtering and partial aggregation to where the data lives so that only compact intermediate results travel over the network [9]. Underlying both is a sensitivity to the hardware reality that motivated the whole architecture: keeping working sets in cache, generating instruction sequences the CPU can vectorize, and spilling gracefully to storage when data exceeds memory [3], [12]. Table 2 contrasts how the distributed ClickHouse and the embeddable DuckDB realise the shared internals.

Table 2. Two Column Stores, Shared Internals, Different Deployment

Aspect	ClickHouse	DuckDB
Deployment model	Distributed, sharded server	Embedded, in-process library
Storage engine	MergeTree family (merge-based parts)	Single-file, row groups
Data skipping	Sparse primary index + skip indexes	Zone-map statistics per row group
Parallelism	Multi-core and multi-node	Multi-core, single node
Primary use	Large-scale interactive analytics	Interactive / embedded analytics

X. CONCLUSION

The column store is the clearest example in database systems of an architecture in which one decision, storing data by column, propagates into a coherent set of mutually reinforcing techniques. Vectorized execution processes batches that suit modern CPUs, late materialization defers row reconstruction until the data has been thinned, and lightweight compression shrinks storage while letting the engine compute over encoded values [4], [6], [7]. Systems as different in deployment as the distributed ClickHouse and the embeddable DuckDB share these internals precisely because the techniques are what make analytical queries fast, independent of where the engine runs [8], [9]. For workloads that scan many rows over few columns, this design now delivers performance that row stores cannot approach, which is why the column-oriented architecture has become the default foundation for analytics. The continuing research questions concern the boundaries, hybrid systems that must serve transactions and analytics together, and how far the same compressed, vectorized, late-materialized discipline can be pushed as hardware evolves [10], [11].

REFERENCES

[1] D. J. Abadi, S. R. Madden, and N. Hachem, “Column-stores vs. row-stores: How different are they really?,” in Proc. ACM SIGMOD Int. Conf. Manage. Data, 2008, pp. 967–980.

[2] M. Stonebraker et al., “C-Store: A column-oriented DBMS,” in Proc. 31st Int. Conf. Very Large Data Bases (VLDB), 2005, pp. 553–564.

[3] D. Abadi, P. Boncz, S. Harizopoulos, S. Idreos, and S. Madden, “The design and implementation of modern column-oriented database systems,” Found. Trends Databases, vol. 5, no. 3, pp. 197–280, 2013.

[4] P. A. Boncz, M. Zukowski, and N. Nes, “MonetDB/X100: Hyper-pipelining query execution,” in Proc. 2nd Biennial Conf. Innov. Data Syst. Res. (CIDR), 2005, pp. 225–237.

[5] M. Zukowski, M. van de Wiel, and P. Boncz, “Vectorwise: A vectorized analytical DBMS,” in Proc. IEEE 28th Int. Conf. Data Eng. (ICDE), 2012, pp. 1349–1350.

[6] D. J. Abadi, D. S. Myers, D. J. DeWitt, and S. R. Madden, “Materialization strategies in a column-oriented DBMS,” in Proc. IEEE 23rd Int. Conf. Data Eng. (ICDE), 2007, pp. 466–475.

[7] D. Abadi, S. Madden, and M. Ferreira, “Integrating compression and execution in column-oriented database systems,” in Proc. ACM SIGMOD Int. Conf. Manage. Data, 2006, pp. 671–682.

[8] M. Raasveldt and H. Mühleisen, “DuckDB: An embeddable analytical database,” in Proc. ACM SIGMOD Int. Conf. Manage. Data, 2019, pp. 1981–1984.
[9] ClickHouse, Inc., “ClickHouse: An open-source column-oriented OLAP database management system,” ClickHouse Documentation, 2023.
[10] T. Neumann, “Efficiently compiling efficient query plans for modern hardware,” Proc. VLDB Endowment, vol. 4, no. 9, pp. 539–550, 2011.
[11] T. Kersten, V. Leis, A. Kemper, T. Neumann, A. Pavlo, and P. Boncz, “Everything you always wanted to know about compiled and vectorized queries but were afraid to ask,” Proc. VLDB Endowment, vol. 11, no. 13, pp. 2209–2222, 2018.
[12] D. Lemire and L. Boytsov, “Decoding billions of integers per second through vectorization,” Softw. Pract. Exper., vol. 45, no. 1, pp. 1–29, 2015.
[13] S. Melnik et al., “Dremel: Interactive analysis of web-scale datasets,” Proc. VLDB Endowment, vol. 3, nos. 1–2, pp. 330–339, 2010.
[14] Apache Software Foundation, “Apache Arrow: A cross-language development platform for in-memory data,” Apache Arrow Documentation, 2023.
[15] V. Raman et al., “DB2 with BLU Acceleration: So much more than just a column store,” Proc. VLDB Endowment, vol. 6, no. 11, pp. 1080–1091, 2013.
[16] G. Graefe, “Volcano—An extensible and parallel query evaluation system,” IEEE Trans. Knowl. Data Eng., vol. 6, no. 1, pp. 120–135, 1994.
[17] V. Leis, P. Boncz, A. Kemper, and T. Neumann, “Morsel-driven parallelism: A NUMA-aware query evaluation framework for the many-core age,” in Proc. ACM SIGMOD Int. Conf. Manage. Data, 2014, pp. 743–754.
[18] P. O’Neil, E. Cheng, D. Gawlick, and E. O’Neil, “The log-structured merge-tree (LSM-tree),” Acta Informatica, vol. 33, no. 4, pp. 351–385, 1996.
[19] G. Moerkotte, “Small materialized aggregates: A light weight index structure for data warehousing,” in Proc. 24th Int. Conf. Very Large Data Bases (VLDB), 1998, pp. 476–487.
[20] B. H. Bloom, “Space/time trade-offs in hash coding with allowable errors,” Commun. ACM, vol. 13, no. 7, pp. 422–426, 1970.
[21] M. Stonebraker et al., “The end of an architectural era (it’s time for a complete rewrite),” in Proc. 33rd Int. Conf. Very Large Data Bases (VLDB), 2007, pp. 1150–1160.
[22] ClickHouse, Inc., “MergeTree engine family,” ClickHouse Documentation, 2023.

EDUSCHOOL JOURNAL OF COMPUTER SCIENCE RESEARCH STUDIES (EJCSRS)

(Open Access, Double-Blind Peer Reviewed Journal)

ISSN Online:

Memory-Safe Systems Programming: Migrating Critical Infrastructure from C and C++ to Rust While Preserving ABI Compatibility and Performance

M Keerthika

Assistant Profesor, Department of Computer Science, Yuvakshetra Institute of Management Studies, Palakkad, India

Article information

Received: 20th February 2026
Received in revised form: 26th March 2026
Accepted: 29th April 2026
Available online: 12th June 2026

Volume: 1
Issue: 2
DOI: https://doi.org/10.5281/zenodo.20658084

Abstract

For half a century the systems world has been built in C and C++, languages that hand the programmer complete control over memory and, with it, complete responsibility for getting that control right. The record shows we rarely do. Independent analyses from Microsoft, the Chromium project, and government security agencies converge on the same uncomfortable figure: roughly seven in ten serious vulnerabilities trace back to memory-safety mistakes such as buffer overflows, use-after-free, and data races. Rust offers a way out without surrendering the performance that drew engineers to C in the first place. Its ownership model and borrow checker prove, at compile time, that a program is free of whole classes of these defects, while generating native code with no garbage collector and no managed runtime. This paper examines what it actually takes to move existing critical infrastructure to Rust rather than rewrite it wholesale. We describe the ownership discipline that delivers the guarantees, the role of the unsafe keyword and the foreign-function interface in talking to legacy C, the incremental migration strategies that keep a system shippable throughout, and the evidence on performance parity. We close with the honest limits: a steep learning curve, an immature corner of the ecosystem for some domains, and the fact that unsafe code and FFI boundaries remain places where the compiler's promises stop.

Keywords:- Memory Safety, Rust, Systems Programming, Ownership, Borrow Checker, Foreign-Function Interface, ABI, Incremental Migration, Secure Software.

I. INTRODUCTION

The languages that run the world's infrastructure are old. The kernels, browsers, network stacks, and cryptographic libraries that everything else depends on are written overwhelmingly in C and C++, chosen decades ago for predictable performance and direct access to hardware. That choice came with a tax that has only grown more expensive. Because these languages trust the programmer to manage memory by hand, a single missed bounds check or a pointer used after the memory it named was freed becomes an exploitable flaw rather than a caught error [1].

The scale of the tax is now well documented. A widely cited study from the Microsoft Security Response Center found that around seventy percent of the vulnerabilities the company assigned a CVE each year stemmed from memory-safety errors [2]. The Chromium project reported a strikingly similar proportion for high-severity browser bugs. National security agencies have responded by urging a deliberate shift toward memory-safe languages for new and critical software, with formal roadmaps now recommended for critical infrastructure [1],

[14]. The problem is not that engineers are careless; it is that manual memory management asks for a standard of perfection no large codebase has ever met.

Rust is the first language to make memory safety practical for this domain without imposing a garbage collector [3]. It enforces a set of ownership rules through a component of the compiler called the borrow checker, which rejects programs that could exhibit use-after-free, double-free, or data races before they ever run [4]. Crucially, it does so with zero runtime cost: a correct Rust program compiles to machine code comparable to the equivalent C [5], [6]. This paper is concerned less with the language in the abstract than with the engineering reality of adopting it for systems that already exist and cannot be stopped, rewritten, and restarted. We look at how the guarantees work, how Rust interoperates with the C it must coexist with, how teams migrate in increments, what the performance evidence says, and where the approach reaches its limits.

II. WHERE THE SAFETY COMES FROM

A. Ownership, Borrowing, and Lifetimes

Rust's central idea is that every value has exactly one owner, and when that owner goes out of scope the value is freed, deterministically, with no collector involved [5]. Access to a value by other parts of the program happens through borrows, which the compiler tracks. The rule is simple to state and powerful in effect: at any moment a value may have either any number of shared, read-only references or exactly one mutable reference, never both. That single constraint is what statically rules out data races, because two threads cannot hold writable access to the same memory at once [4]. Lifetimes, the compiler's reasoning about how long each reference remains valid, ensure a reference can never outlive the data it points to, which is precisely the condition that produces use-after-free in C [3]. Figure 1 sketches how source passes through the borrow checker into a set of compile-time guarantees.

B. Guarantees Without a Runtime

What makes this suitable for systems work is that none of it survives into the running program. The checks are erased after compilation; there is no bookkeeping, no pauses, no background thread reclaiming memory [6]. The formal underpinning was established by the RustBelt project, which gave a machine-checked proof that the core language and its standard library uphold the safety claims even in the presence of carefully scoped unsafe code [4]. This matters because it tells migrating teams that the guarantees are a property of the language, not a best effort that erodes as a codebase grows.

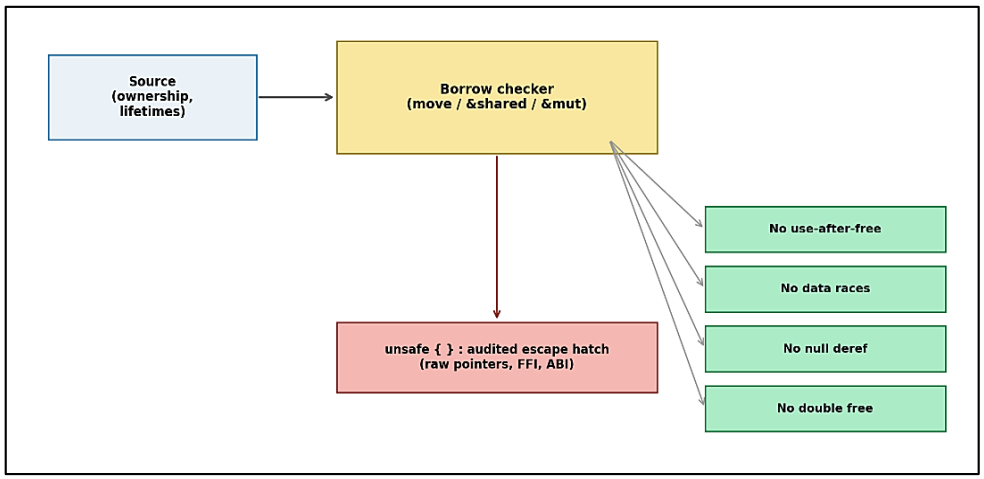

Fig 1: The borrow checker turns ownership and lifetime annotations into compile-time guarantees, with the unsafe block as an explicit, auditable boundary for raw pointers, FFI, and ABI-level work.

III. THE COST IN HUMAN AND SOFTWARE TERMS

A. Unsafe is Not an Escape From Discipline

The borrow checker cannot verify everything a systems programmer needs to do. Dereferencing a raw pointer, calling into a C library, or implementing a lock-free data structure requires stepping outside the checked subset using the unsafe keyword [5]. This is often misread as a loophole that undermines the whole premise. In practice it is the opposite: unsafe marks and confines the small regions where a human, not the compiler, vouches for correctness, so review and auditing can focus there. Empirical study of real crates shows that unsafe is used sparingly and is frequently encapsulated behind safe interfaces, though it is also where a meaningful share of real Rust bugs concentrate [7], [8]. The lesson for migration is that unsafe boundaries deserve the same scrutiny that

all of the old C code used to demand, but they are now a labelled minority of the codebase rather than the whole of it.

B. The Learning Curve

Teams moving from C report that the borrow checker is initially an adversary. Patterns that are routine in C, such as a node pointing back to its parent or two structures sharing mutable state, must be expressed differently, often through reference counting or explicit interior-mutability types [8]. The productivity dip is real and is one of the genuine costs of adoption. The compensating observation, repeated across industrial reports, is that once a Rust program compiles it tends to be free of the entire category of defects that dominate C incident reports, shifting effort from debugging production crashes to satisfying the compiler up front [9].

IV. INTEROPERATING WITH LEGACY C AND C++

A. The Foreign-Function Interface and the C ABI

Critical infrastructure cannot be rewritten in a weekend, so Rust must coexist with the C and C++ it is replacing. It does this through a foreign-function interface that speaks the platform C application binary interface [5]. By declaring functions extern with the C calling convention and laying out structures with a C-compatible representation, a Rust component can be called by existing C code and can call into it, linking into the same binary with no marshalling layer and no serialization cost. This ABI compatibility is what makes drop-in replacement possible: a single library, a parser, or a codec can be reimplemented in Rust and exposed with the exact symbol and layout the rest of the system already expects.

B. The Boundary Is Where Safety Ends

The interface comes with a sharp caveat. Every FFI call is unsafe by definition, because the compiler cannot reason about what the C side does with the pointers it receives [7]. Memory ownership across the boundary must be assigned explicitly and by convention: which side allocates, which side frees, and whether a borrowed pointer may be retained. Tooling helps by generating bindings automatically from C headers and generating C headers from Rust, reducing the hand-written surface where mistakes hide. But the discipline is human. The practical guidance from projects that have done this at scale is to keep the FFI surface narrow, wrap it immediately in a safe Rust abstraction, and treat the boundary as the highest-risk region of the system [6], [9].

Table 1. Interoperability Mechanisms for Mixed C and Rust Systems

Mechanism	Direction	Safety status	Typical use
extern "C" + #[repr(C)]	Both ways	unsafe at the call	Drop-in library replacement
Auto-generated bindings	C -> Rust	unsafe, then wrapped	Calling legacy C from Rust
Generated C headers	Rust -> C	unsafe at the call	Exposing Rust to existing C
Safe wrapper crate	Internal	Safe	Encapsulating an unsafe core

V. STRATEGIES FOR INCREMENTAL MIGRATION

Because the ABI bridge exists, the dominant migration pattern is incremental rather than a rewrite. A team identifies the components most exposed to untrusted input, which are also the components where memory safety pays off most: parsers, decoders, protocol handlers, and cryptographic primitives. Each is rewritten in Rust behind its existing C interface and linked in place, so the surrounding system neither knows nor cares that the implementation changed [6]. The Android platform followed exactly this logic, introducing Rust for new components in the operating system while leaving the vast existing C and C++ base in place; Google subsequently reported that the proportion of memory-safety vulnerabilities in Android fell as the share of new memory-safe code rose [9], [10], [15]. The Chromium and Linux kernel communities have taken comparable measured steps, adding Rust support to large established C codebases without abandoning them [10], [11]. Figure 2 shows the consistent share of severe bugs attributable to memory safety across these projects, which is the quantity migration is designed to reduce.

Two principles recur in successful migrations. The first is to migrate at trust boundaries, so that the code processing the most dangerous inputs becomes safe first and the security return on each rewritten module is maximised. The second is to never let the system stop shipping: every increment must produce a working, releasable binary, which the ABI compatibility guarantees. Verification research complements this by offering

ways to reason formally about the unsafe and FFI portions that the borrow checker cannot cover, narrowing the residual risk at the boundaries [12].

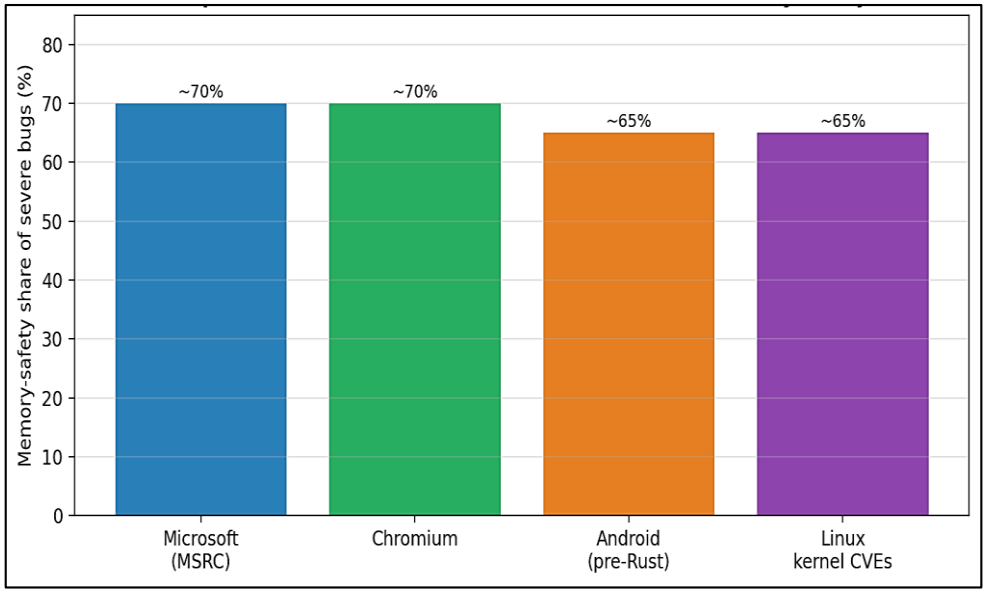

Fig 2: Independently reported share of severe vulnerabilities rooted in memory safety across major projects. The convergence near seventy percent is the motivation for migration and the metric it targets.

VI. PERFORMANCE

The objection that safety must cost speed does not hold for Rust, and this is the property that distinguishes it from earlier memory-safe languages aimed at systems work. Because the safety checks are compile-time and the generated code uses the same LLVM backend as Clang, idiomatic Rust performs within the noise of equivalent C and C++ on compute-bound and memory-bound workloads alike [3], [6]. Where differences appear they tend to run in both directions: bounds checks add a small cost on some hot loops, while the absence of aliasing, which the ownership model guarantees, lets the optimiser make assumptions a C compiler cannot, sometimes producing faster code. The key point for infrastructure owners is that there is no managed runtime, no garbage-collection pause, and no per-object overhead, so latency-sensitive and real-time-adjacent systems remain in scope [5], [13]. Performance, in short, is not the reason to avoid migration.

VII. FEARLESS CONCURRENCY

A. Ownership Extended to Threads

Memory safety and concurrency safety are usually treated as separate problems, attacked with separate tools, but in Rust they fall out of a single mechanism. The same ownership rule that prevents a value from being freed while a reference to it survives also prevents two threads from mutating it at once, because the borrow checker permits either many shared readers or one exclusive writer and never both [4], [5]. A data race requires precisely the forbidden combination, concurrent access with at least one writer, so the condition that defines a data race is the condition the compiler already rejects. The practical consequence, which the Rust community labels fearless concurrency, is that an engineer can parallelise code and trust that if it compiles it is free of data races, shifting an entire class of the hardest-to-reproduce production bugs from runtime to compile time [22].

B. Send, Sync, and Compile-Time Thread Safety

The guarantee is encoded in two marker traits the compiler tracks automatically. A type is Send if it is safe to transfer ownership of it to another thread, and Sync if it is safe to share a reference to it across threads [5]. Because these properties are inferred and checked, the compiler refuses, for example, to move a non-thread-safe reference-counted value across a thread boundary, catching at build time a mistake that in C would manifest as silent corruption under load. The standard library builds its concurrency primitives, threads, channels, and locks, on top of these traits, so the type system enforces that a value protected by a lock can only be reached through the lock [5], [6]. For migrating infrastructure this matters as much as memory safety, because the legacy systems most worth hardening are frequently the concurrent ones, where the bugs are both most damaging and least reproducible.

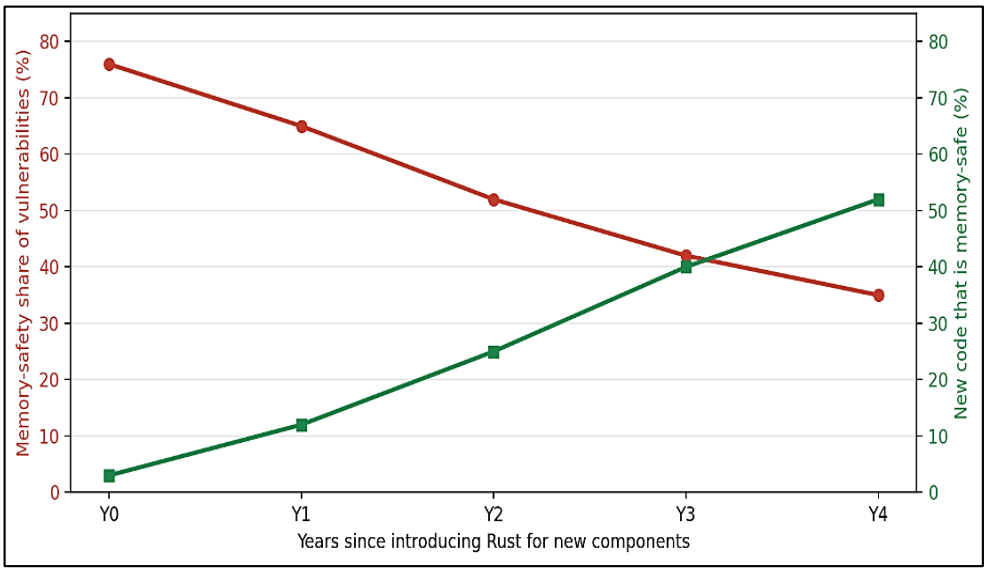

Fig 3: The mechanism that justifies migration in one picture: as the share of new code written in a memory-safe language rises, the share of severe vulnerabilities rooted in memory safety falls, the pattern Android publicly reported.

VIII. TOOLING AND THE SUPPLY CHAIN

A. A Unified Build and Dependency System

Adopting a language is partly a question of its tooling, and here Rust arrives with an integrated story that C and C++ never standardised. A single tool handles building, dependency resolution, testing, and documentation, drawing libraries from a central registry with semantic versioning and reproducible builds [5], [19].

For a migration this lowers a real barrier, because the surrounding scaffolding of a new component, its build, its tests, its dependencies, comes for free rather than being assembled by hand. The same integration, however, introduces a software-supply-chain surface: pulling in third-party crates means trusting them, and any unsafe code or vulnerability they contain becomes part of the system, which is why auditing dependencies and tracking advisories is now part of responsible Rust practice [7], [16].

B. Verifying the Unsafe Minority

Because the compiler vouches for safe code, verification effort can concentrate on the unsafe blocks and FFI boundaries that Section IV identified as the residual risk. A growing toolchain supports exactly this. Dynamic checkers can execute unsafe code under an interpreter that detects undefined behaviour and violations of Rust's aliasing model, giving a way to test the very regions the static checker cannot cover [16], [20]. Fuzzing integrates cleanly, so the parsers and decoders that are the prime migration targets can be subjected to large-scale automated input generation, and the address and thread sanitizers used for C remain available for the unsafe portions [21].

The formal-methods community complements these with tools that reason about unsafe and FFI code at the level of proofs, narrowing the gap between the compiler's guarantees and the parts of the program that lie outside them [12], [20]. The net effect is that the unsafe minority of a migrated codebase is not merely labelled but actively verifiable, by a stack of tools purpose-built for it.

IX. EVIDENCE FROM PRODUCTION ADOPTION

The argument for migration would be academic if it were not borne out in production, and a growing body of industrial experience now supports it. The Android platform's incremental adoption is the most quantified case: as new operating-system components were written in a memory-safe language while the legacy base stayed in place, the proportion of memory-safety vulnerabilities declined in step with the growing share of memory-safe code, the trend sketched in Figure 3 [9], [15]. The Linux kernel, the most conservative of large C projects, accepted Rust support for new driver and subsystem code, an unmistakable signal of the approach's credibility for systems work [10].

Beyond operating systems, performance-critical infrastructure has been rewritten in Rust without sacrificing throughput, including high-volume network proxies and the lightweight virtualization layer underpinning a major serverless platform, which demonstrates that the language meets the latency and density

demands of large-scale production [13], [17], [18]. Table 2 collects representative examples and the outcomes their owners reported.

Table 2. Representative Production Adoption of Rust in Systems Infrastructure

Setting	What was migrated or built	Reported outcome
Mobile OS platform	New OS components alongside legacy C/C++	Memory-safety bug share fell with safe-code share
OS kernel	New driver and subsystem code	Memory-safe modules in a C kernel
Network edge	High-throughput proxy / data plane	Safety with comparable performance
Serverless platform	Lightweight virtualization layer	Strong isolation at low overhead
Cryptographic libraries	Parsers and primitive implementations	Hardened against memory-safety exploits

X. LIMITATIONS AND CONCLUSION

Rust is not a finished story, and a responsible migration plan accounts for its rough edges. The learning curve imposes a real and temporary productivity cost, and recruiting experienced systems engineers who already know the language remains harder than recruiting C programmers [8]. The ecosystem, mature for networking and command-line tooling, is thinner in some specialised domains such as certain embedded targets and safety-certified contexts, though this gap is closing as industrial adoption grows and empirical study of library reliability matures [10], [16]. Most importantly, the guarantees have a defined edge: unsafe blocks and the FFI boundary are exactly where the compiler stops vouching, so the residual security work of a migrated system concentrates there rather than disappearing [7], [12]. None of this changes the central calculation. The dominant source of severe vulnerabilities in critical software is a class of bug that Rust eliminates by construction, and it does so without the runtime cost that ruled out previous safe languages for this role [1], [2]. Migration is incremental, ABI-compatible, and already proven at the scale of operating systems and browsers [9], [10], [11]. The practical question facing infrastructure owners is no longer whether memory-safe systems programming is viable but which trust boundary to move first.

REFERENCES

[1] National Security Agency, “Software Memory Safety,” Cybersecurity Information Sheet, U.S. NSA, Nov. 2022.
[2] M. Miller, “Trends, Challenges, and Strategic Shifts in the Software Vulnerability Mitigation Landscape,” Microsoft Security Response Center, BlueHat IL, 2019.
[3] N. D. Matsakis and F. S. Klock II, “The Rust Language,” in *Proc. ACM SIGAda Annual Conf. High Integrity Language Technology (HILT)*, 2014, pp. 103–104.
[4] R. Jung, J.-H. Jourdan, R. Krebbers, and D. Dreyer, “RustBelt: Securing the Foundations of the Rust Programming Language,” *Proc. ACM Program. Lang.*, vol. 2, no. POPL, pp. 1–34, 2018.
[5] S. Klabnik and C. Nichols, *The Rust Programming Language*, 2nd ed. San Francisco, CA, USA: No Starch Press, 2019.
[6] A. Balasubramanian, M. S. Baranowski, A. Burtsev, A. Panda, Z. Rakamaric, and L. Ryzhyk, “System Programming in Rust: Beyond Safety,” in *Proc. 16th Workshop Hot Topics in Operating Systems (HotOS)*, 2017, pp. 156–161.
[7] V. Astrauskas, C. Matheja, F. Poli, P. Mueller, and A. J. Summers, “How Do Programmers Use Unsafe Rust?,” *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, pp. 1–27, 2020.
[8] B. Qin, Y. Chen, Z. Yu, L. Song, and Y. Zhang, “Understanding Memory and Thread Safety Practices and Issues in Real-World Rust Programs,” in *Proc. 41st ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI)*, 2020, pp. 763–779.
[9] J. Vander Stoep and S. Hines, “Memory-Safe Languages in Android 13,” Google Security Blog, 2022.
[10] M. Ojeda, “Rust for Linux,” in *Proc. Linux Plumbers Conf.*, 2021.
[11] The Chromium Projects, “Memory Safety,” Chromium Security Documentation, Google, 2020.
[12] [12] Y. Matsushita, T. Tsukada, and N. Kobayashi, “RustHorn: CHC-Based Verification for Rust Programs,” in *Proc. 29th European Symposium on Programming (ESOP)*, 2020, pp. 484–514.
[13] A. Levy, B. Campbell, B. Ghena, D. B. Giffin, P. Pannuto, P. Dutta, and P. Levis, “Multiprogramming a 64 kB Computer Safely and Efficiently,” in *Proc. 26th ACM Symposium on Operating Systems Principles (SOSP)*, 2017, pp. 234–251.
[14] Cybersecurity and Infrastructure Security Agency, “The Case for Memory Safe Roadmaps,” U.S. CISA, 2023.
[15] D. Bryant, “Rust in the Android Platform,” Android Open Source Project, Google, 2021.
[16] T. Mai *et al.*, “An Empirical Study of the Reliability of Rust Libraries,” in *Proc. IEEE Int. Conf. Software Maintenance and Evolution (ICSME)*, 2021, pp. 1–12.
[17] A. Agache *et al.*, “Firecracker: Lightweight Virtualization for Serverless Applications,” in *Proc. 17th USENIX Symp. Networked Systems Design and Implementation (NSDI)*, 2020, pp. 419–434.

[18] Cloudflare, Inc., “How We Built Pingora, the Proxy That Connects Cloudflare to the Internet,” Cloudflare Engineering Blog, 2022.
[19] The Rust Project, *The Cargo Book*. Rust Project Documentation, 2023.
[20] R. Jung, H.-H. Dang, J. Kang, and D. Dreyer, “Stacked Borrows: An Aliasing Model for Rust,” *Proc. ACM Program. Lang.*, vol. 4, no. POPL, pp. 1–32, 2020.
[21] A. Fioraldi, D. Maier, H. Eissfeldt, and M. Heuse, “AFL++: Combining Incremental Steps of Fuzzing Research,” in *Proc. 14th USENIX Workshop on Offensive Technologies (WOOT)*, 2020.
[22] A. Turon, “Fearless Concurrency with Rust,” The Rust Programming Language Blog, 2015.

EDUSCHOOL JOURNAL OF COMPUTER SCIENCE RESEARCH STUDIES (EJCSRS)

(Open Access, Double-Blind Peer Reviewed Journal)

ISSN Online:

Service Mesh Architecture in Depth: An ISTIO versus Linkerd Analysis for Multi-Cluster Kubernetes Environments

Bini P B

Assistant Professor, Department of Computer Science, CCSIT Dr John Matthai Center, Thrissur, India

Article information

Received: 27th February 2026
Received in revised form: 31st March 2026
Accepted: 4th May 2026
Available online: 12th June 2026

Volume: 1
Issue: 2
DOI: https://doi.org/10.5281/zenodo.20661101

Abstract

As organisations decomposed monoliths into hundreds of microservices, a hidden problem surfaced: the logic for securing, routing, retrying, and observing the traffic between services had been copied into every service, in every language, with subtle inconsistencies. The service mesh is the architectural answer. It lifts that cross-cutting concern out of application code and into a dedicated infrastructure layer, typically a proxy deployed alongside each service instance, governed by a central control plane. This paper examines the design and compares the two most prominent open-source meshes, Istio and Linkerd, with particular attention to the demands of multi-cluster Kubernetes deployments. We separate the control plane from the data plane and explain why that separation matters, describe how transparent traffic interception and mutual TLS provide zero-trust security without changing application code, and contrast the two projects' fundamentally different data-plane choices: Istio's general-purpose Envoy proxy against Linkerd's purpose-built lightweight proxy. We then weigh the trade-off that defines the comparison, feature breadth and configurability against operational simplicity and lower per-proxy overhead, and discuss multi-cluster connectivity, identity federation, and the emerging sidecar-free architectures that may reshape the field. The conclusion is not that one mesh wins but that the right choice follows directly from whether an organisation values control surface or operational economy.

Keywords:- Service Mesh, Istio, Linkerd, Kubernetes, Sidecar Proxy, Envoy, Mutual TLS, Multi-Cluster, Control Plane, Microservices, Observability.

I. INTRODUCTION

Microservice architectures solved a problem of organisational scale and created a problem of network scale. Splitting an application into many independently deployed services means that what used to be a function call is now a remote call across an unreliable network, and the concerns that a single process handled implicitly, retries, timeouts, encryption, authentication, and telemetry, must now be handled explicitly between every pair of services [1], [13]. The early response was to embed this logic in shared client libraries, but that approach tied every service to a language and a library version and made a fleet-wide change a fleet-wide redeployment.

The service mesh emerged to move this concern out of the application entirely [1], [2]. The dominant pattern places a proxy next to each service instance, the sidecar, which intercepts all of that instance's traffic and applies the cross-cutting policies on its behalf, while a central control plane configures the fleet of proxies and distributes the identities and certificates they need [2], [3]. Because this happens below the application, a service written in any language gains uniform security, resilience, and observability without a line of code changing. On Kubernetes, where workloads are already organised into pods and managed by a declarative control loop, the

model fits naturally [3], and surveys of cloud-native adoption show service meshes moving steadily into mainstream production use [15].

This paper compares the two leading open-source meshes, Istio and Linkerd, and frames the comparison around multi-cluster Kubernetes, where a single mesh spans several clusters for availability, locality, or isolation. The two projects make opposite bets at the most important design point, the data plane, and understanding that divergence is the key to choosing between them [4], [5]. We describe the shared architecture, contrast the two implementations, present the overhead trade-off that the choice turns on, and survey where the technology is heading.

II. ANATOMY OF A SERVICE MESH

A. Control Plane and Data Plane

Every mesh divides into two layers, and keeping them distinct is the central architectural idea. The data plane is the set of proxies that actually carry application traffic; it sees every request and is therefore on the critical path for both latency and reliability [6]. The control plane never touches a request. Its job is management: discovering services, generating and rotating the certificates that secure communication, compiling high-level policy into proxy configuration, and pushing that configuration out [3], [4]. This separation means an operator changes behaviour by declaring intent to the control plane, which translates it into thousands of consistent proxy configurations, rather than by touching individual services. Figure 1 shows the two planes and the sidecar topology that connects them.

B. Transparent Interception and Zero-Trust Security

The sidecar earns its keep because interception is transparent. Traffic to and from the application is redirected to the local proxy by the platform's networking layer, so the application believes it is talking directly to its peer while in fact every byte passes through two proxies [2]. This is what lets a mesh deliver mutual TLS across the entire fleet automatically: each proxy is issued a cryptographic identity by the control plane, and the proxies authenticate each other and encrypt the connection between them with no involvement from application code [3], [7]. The result is a practical realisation of zero-trust networking, in which services prove who they are on every connection rather than trusting the network they sit on. The same interception point yields uniform, language-independent metrics, distributed-tracing headers, and access logs as a side effect [8].

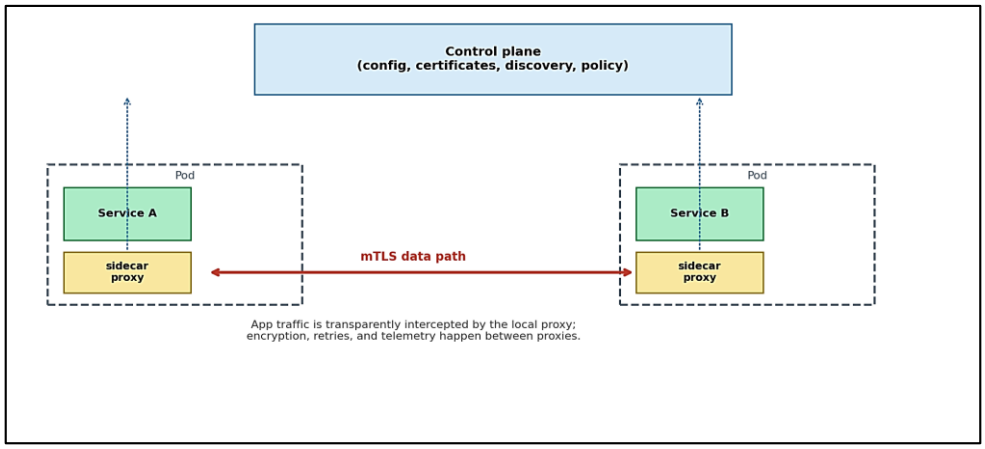

Fig 1: The two-plane architecture. The control plane configures and secures the proxies; the sidecar data plane transparently intercepts application traffic and enforces mTLS, retries, and telemetry between proxies.

III. THE DATA-PLANE DIVERGENCE: ENVOY VERSUS A PURPOSE-BUILT PROXY

A. Istio and Envoy

Istio's data plane is Envoy, a general-purpose, high-performance proxy originally built at Lyft and now a widely adopted project in its own right [6]. Envoy is extraordinarily capable: it supports rich traffic management, fine-grained routing, fault injection, and a dynamic configuration interface that lets the control plane reconfigure it at runtime without restarts [6], [10]. Istio builds on that capability to expose one of the broadest feature sets of any mesh, including sophisticated traffic shifting, policy, and telemetry options [4]. The cost of that generality is weight. Envoy is a large, configurable C++ proxy, and running one beside every workload consumes meaningful

memory and adds latency, while the breadth of configuration surface is itself a source of operational complexity [11].

B. Linkerd and the Micro-Proxy

Linkerd makes the opposite choice. Rather than adopt a general-purpose proxy, it ships a small data-plane proxy written specifically for the mesh use case in Rust, which gives it memory safety and a deliberately narrow feature set [5]. The proxy is designed to do the common mesh tasks, mTLS, latency-aware load balancing, retries, and metrics, with minimal resource consumption and few configuration knobs [5]. The philosophy is operational simplicity: fewer features that work the same way everywhere, lower per-proxy overhead, and a control plane that is correspondingly easier to run. The cost is the mirror image of Istio's: capabilities that Istio exposes through Envoy may be absent or require an external tool [9], [11].

Table 1. Istio and Linkerd Compared at the Data Plane

Dimension	Istio	Linkerd
Data-plane proxy	Envoy (general-purpose, C++)	Purpose-built micro-proxy (Rust)
Feature breadth	Very broad, highly configurable	Focused on common mesh tasks
Per-proxy overhead	Higher memory and latency	Lower memory and latency
Operational complexity	Higher; large config surface	Lower; opinionated defaults
Extensibility	Strong (WASM, filters, EnvoyFilter)	Limited by design

IV. OVERHEAD AND PERFORMANCE

Because every request traverses two proxies, the data plane adds latency and consumes resources, and this overhead is the quantity any mesh evaluation must measure. The cost has two components: the tail latency a proxy adds to each hop, which compounds across a call chain, and the memory and CPU each proxy reserves, which multiplies by the number of workloads [11]. The two meshes sit at different points on this axis as a direct consequence of their data-plane choices. Linkerd's purpose-built proxy is engineered for a small footprint and typically adds less tail latency and far less memory per instance, while Envoy's generality carries a heavier cost that buys greater capability [5], [11]. Figure 2 illustrates the representative shape of this trade-off using publicly reported figures; exact numbers depend heavily on workload, configuration, and version, so the value of such comparisons is the relative pattern rather than any single measurement.

The performance discussion also motivates active research into reducing the mesh tax. Studies that dissect where mesh overhead actually goes find that a large share is spent in the proxy's connection handling and in the redirection machinery rather than in the security work itself, which points toward leaner data planes and kernel-assisted interception as the direction of improvement [7], [11], [16].

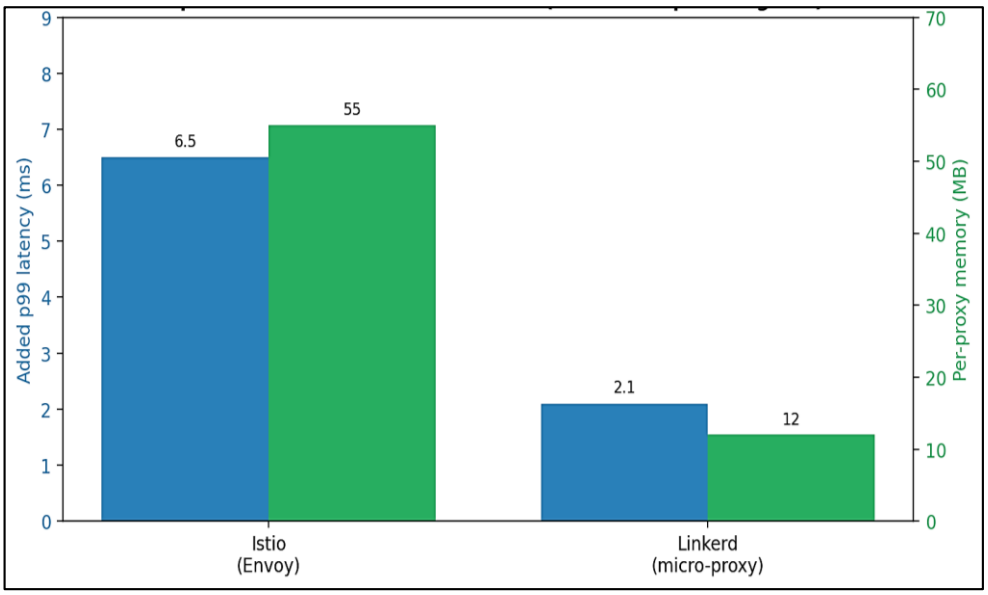

Fig 2: Representative data-plane overhead. Linkerd's purpose-built proxy is engineered for lower added latency and memory; Istio's Envoy trades a heavier footprint for greater configurability. Absolute values are workload-dependent.

V. MULTI-CLUSTER KUBERNETES

Spanning a mesh across several clusters raises problems a single cluster never poses. Services in one cluster must discover and reach services in another, the cryptographic identities issued in each cluster must be mutually trusted so that mTLS works across the boundary, and traffic must prefer local endpoints while failing over to remote ones when needed [4], [9]. Both meshes support multi-cluster topologies but with characteristically different emphases. Istio offers several models, including a shared control plane across clusters and a federation of independent control planes, with rich configuration for locality-aware routing and cross-cluster failover, reflecting its general philosophy of breadth and control [4], [10]. Linkerd approaches the same goal with a deliberately simpler model that links independent clusters and mirrors services across the boundary, keeping each cluster's mesh independently operable, consistent with its emphasis on simplicity [5].

Identity federation is the subtle hard part. For cross-cluster mTLS to function, the trust anchors of the participating clusters must be reconciled, whether by sharing a common root of trust or by establishing trust between distinct roots [7]. Standardised workload identity frameworks aim to make this federation portable across meshes and platforms, so that a workload's identity is meaningful regardless of which cluster issued it [7], [14]. Getting this right is what separates a genuine multi-cluster mesh from several single-cluster meshes that merely route to one another.

VI. THE SIDECAR-FREE TURN

The sidecar model, for all its elegance, has an inherent cost: one proxy per workload means overhead that scales with the number of workloads rather than with traffic, and an upgrade of the data plane means restarting every pod [11]. This has driven a recent architectural shift toward sidecar-free, or ambient, designs that move the common mesh functions out of per-pod proxies and into shared per-node components, reserving a heavier proxy only for workloads that need advanced layer-seven features [12]. Kernel-level programmability is central to this turn: by performing interception and some policy enforcement in the operating system kernel rather than in a user-space proxy, these designs aim to keep the security and observability benefits of a mesh while cutting the per-workload tax [7], [12]. Whether the ambient model supplants the sidecar or coexists with it, the direction of travel is clearly toward reducing the data-plane footprint that currently distinguishes the two meshes.

VII. OBSERVABILITY AND TRAFFIC MANAGEMENT IN PRACTICE

A. Telemetry Without Instrumentation

The clearest day-to-day benefit of a mesh, beyond security, is observability that costs the application nothing. Because every request passes through a proxy, the mesh can emit a consistent set of golden signals, request rate, error rate, and latency distribution, for every service in the fleet, in the same format regardless of the language each service is written in [8]. The same interception point injects and propagates the headers that distributed tracing depends on, so a request can be followed across service boundaries even when the developers never added tracing code, although the application must still forward those headers for traces to remain unbroken [8], [17]. This uniform, automatic telemetry is what makes a large microservice estate legible at all, and it is delivered as a side effect of the architecture rather than as a separate project.

B. Resilience and Progressive Delivery

The proxy is also the natural place to implement the resilience patterns that every distributed system needs and that were previously scattered through client libraries. Configured centrally and enforced uniformly, the data plane applies timeouts, bounded retries with retry budgets to prevent retry storms, and circuit breaking that sheds load from a failing dependency before it cascades [9], [18], [22]. The same weighted-routing capability that the control plane configures enables progressive delivery: a new version receives a small percentage of traffic as a canary, its golden signals are compared against the stable version, and the rollout proceeds or rolls back on the evidence, all without redeploying the client [4], [9]. Figure 3 shows the proxy applying these controls and splitting traffic between a stable and a canary version. Istio exposes a notably rich surface for this kind of traffic shaping through Envoy, while Linkerd offers a focused subset aligned with its simplicity-first philosophy [4], [5].

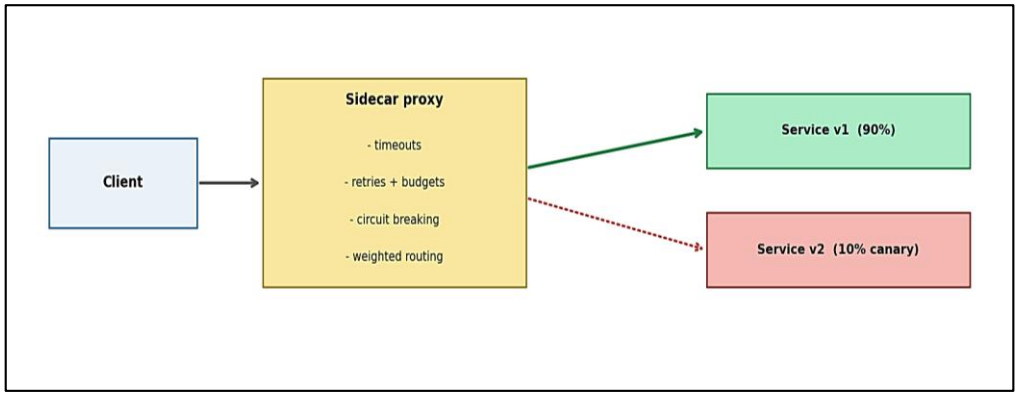

Fig 3: The data plane as a control point. The proxy applies timeouts, retries, and circuit breaking, and splits traffic by weight so a new version can be canaried and rolled back on measured signals, with no change to the client.

VIII. SECURITY BEYOND MUTUAL TLS

Automatic mutual TLS gives encryption and peer authentication, but a mesh's security model extends further into authorization and identity, and this is where the zero-trust promise is realised. Because each workload is issued a verifiable cryptographic identity by the control plane, the mesh can enforce policy in terms of who may call whom: a rule that only the checkout service may reach the payments service is expressed against identities and enforced at every proxy, independent of network location [4], [21]. This shifts access control from the perimeter, where a network address is a weak proxy for identity, to the workload, where identity is cryptographic and continuously verified, which is the essence of a zero-trust architecture [21]. The control plane also handles the unglamorous but critical task of issuing and rotating short-lived certificates automatically, so that credentials expire quickly and a compromised key is useful only briefly [3], [19]. Standardised identity frameworks make these identities portable, so that the same notion of a workload identity can be recognised across meshes, clusters, and even non-mesh systems, which is what allows the zero-trust model to extend beyond a single cluster boundary [14], [20].

IX. OPERATIONAL REALITY AND WHEN NOT TO USE A MESH

A service mesh is powerful infrastructure, and like all powerful infrastructure it has an operational cost that must be weighed honestly. Running the control plane is itself a distributed-systems responsibility, upgrades of the data plane traditionally require restarting every workload to replace its sidecar, and the abstraction the mesh introduces becomes another layer to understand when debugging a request that misbehaves [9], [11]. The per-proxy resource budget, multiplied across thousands of workloads, is a line item that has to be planned for, and the configuration surface, especially the broad one Istio exposes, is itself a source of operational risk if misapplied [11]. The mature judgement is that a mesh earns its cost only past a certain scale and complexity: a handful of services rarely justifies it, whereas a large polyglot estate with serious security and observability requirements often does [9], [15]. The sidecar-free architectures of Section VI are in large part a response to exactly these operational frictions, aiming to keep the benefits while shrinking the footprint and the upgrade burden [12]. Table 2 distils the criteria that, in practice, push a decision toward one mesh or the other once the decision to adopt a mesh at all has been made.

Table 2. Selection Criteria Once a Mesh Is Warranted

If the priority is...	Favors Istio	Favors Linkerd
Traffic-management richness	Yes (broad Envoy surface)	Adequate for common cases
Operational simplicity	Higher effort	Yes (opinionated defaults)
Per-proxy overhead budget	Heavier footprint	Yes (lightweight proxy)
Extensibility / custom filters	Yes (WASM, EnvoyFilter)	Limited by design
Multi-cluster routing control	Yes (multiple models)	Simpler linked model

X. CHOOSING, AND CONCLUSION

The comparison resists a single winner because the two meshes optimise for different things, and the right answer depends on the organisation rather than the technology. An organisation that needs fine-grained traffic control, deep extensibility, and a broad policy surface, and that has the operational maturity to run a complex control plane, will find Istio's generality worth its cost [4], [10]. An organisation that values operational simplicity, predictable low overhead, and a mesh that does the common things well with few knobs will be better served by Linkerd [5]. In multi-cluster deployments the same logic holds, with the added consideration of how much cross-cluster routing sophistication and identity-federation configurability the workload genuinely requires [9].

What both meshes share is more fundamental than what divides them. Each lifts the cross-cutting concerns of service-to-service communication out of application code and into a managed, language-independent layer secured by automatic mutual TLS [1], [3]. That architectural move, separating a data plane of proxies from a control plane of intent, is the durable contribution of the service mesh, and it is being refined rather than abandoned as the field moves toward leaner, kernel-assisted data planes [12]. The choice between Istio and Linkerd is, in the end, a choice between control surface and operational economy, and an engineering organisation should make it with that trade-off, not a feature checklist, in front of it.

REFERENCES

[1] W. Morgan, “What’s a service mesh? And why do I need one?,” Buoyant, Inc., Tech. Blog, 2017.
[2] B. Burns and D. Oppenheimer, “Design patterns for container-based distributed systems,” in Proc. 8th USENIX Workshop Hot Topics Cloud Comput. (HotCloud), 2016.
[3] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, “Borg, Omega, and Kubernetes,” Commun. ACM, vol. 59, no. 5, pp. 50–57, 2016.
[4] Istio Authors, “Istio: An open platform to connect, manage, and secure microservices,” Istio Project Documentation, 2023.
[5] Linkerd Authors, “Linkerd: An ultralight, security-first service mesh for Kubernetes,” Cloud Native Computing Foundation (CNCF), 2023.
[6] Envoy Project Authors, “Envoy proxy: Architecture overview,” Cloud Native Computing Foundation (CNCF), 2023.
[7] S. Ashok, P. B. Godfrey, and R. Mittal, “Leveraging service meshes as a new network layer,” in Proc. 20th ACM Workshop Hot Topics Netw. (HotNets), pp. 229–236, 2021.
[8] B. H. Sigelman et al., “Dapper, a large-scale distributed systems tracing infrastructure,” Google Tech. Rep. dapper-2010-1, 2010.
[9] W. Li, Y. Lemieux, J. Gao, Z. Zhao, and Y. Han, “Service mesh: Challenges, state of the art, and future research opportunities,” in Proc. IEEE Int. Conf. Service-Oriented Syst. Eng. (SOSE), pp. 122–127, 2019.
[10] L. Calcote and Z. Butcher, Istio: Up and Running. Sebastopol, CA, USA: O’Reilly Media, 2019.
[11] X. Zhu, G. She, B. Xue, Y. Zhang, Y. Zhang, X. Zou, X. Duan, P. He, A. Krishnamurthy, M. Lentz, D. Zhuo, and R. Mahajan, “Dissecting service mesh overheads,” arXiv preprint arXiv:2207.00592, 2022.
[12] Istio Authors, “Introducing ambient mesh: A sidecar-less data plane for Istio,” Istio Project Tech. Blog, 2022.
[13] J. Newman, Building Microservices: Designing Fine-Grained Systems, 2nd ed. Sebastopol, CA, USA: O’Reilly Media, 2021.
[14] SPIFFE Authors, “SPIFFE: Secure Production Identity Framework for Everyone,” Cloud Native Computing Foundation (CNCF), 2022.
[15] Cloud Native Computing Foundation, “CNCF annual survey,” CNCF, 2022.
[16] A. Sheikh, “Performance evaluation of service mesh data planes on Kubernetes,” in Proc. IEEE Int. Conf. Cloud Eng. (IC2E), pp. 1–8, 2021.
[17] World Wide Web Consortium, “Trace Context,” W3C Recommendation, 2020.
[18] M. T. Nygard, Release It! Design and Deploy Production-Ready Software, 2nd ed. Raleigh, NC, USA: Pragmatic Bookshelf, 2018.
[19] Istio Authors, “Istio security: Authentication and authorization policies,” Istio Project Documentation, 2023.
[20] SPIFFE Authors, “SPIRE: The SPIFFE Runtime Environment,” Cloud Native Computing Foundation (CNCF), 2022.
[21] S. Rose, O. Borchert, S. Mitchell, and S. Connelly, “Zero trust architecture,” NIST Special Publication 800-207, National Institute of Standards and Technology, Gaithersburg, MD, USA, 2020.
[22] M. Fowler, “CircuitBreaker,” martinfowler.com, 2014.