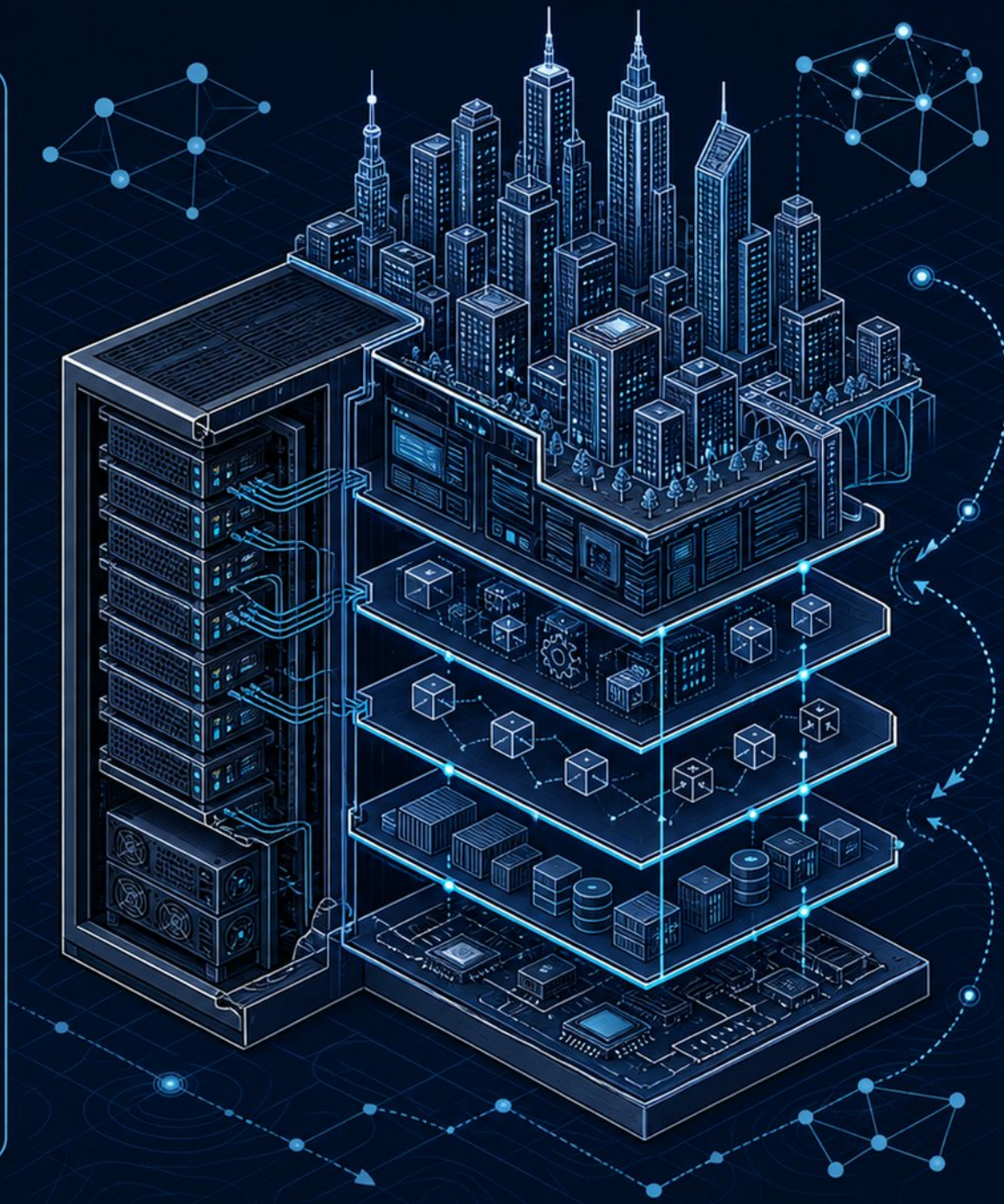
Introduction to Distributed Systems

- - A collection of independent computers that appears as a single coherent system”
- - Why distribution: scalability, fault tolerance, shared resources, latency hiding”
- - Everyday examples: web search, banking, games, streaming, ride-sharing”
- - Core challenges: concurrency, no global clock, independent failures”
- - 2026 context: backbone of AI grids, edge computing, federated orchestration”

System Models and Communication Paradigms

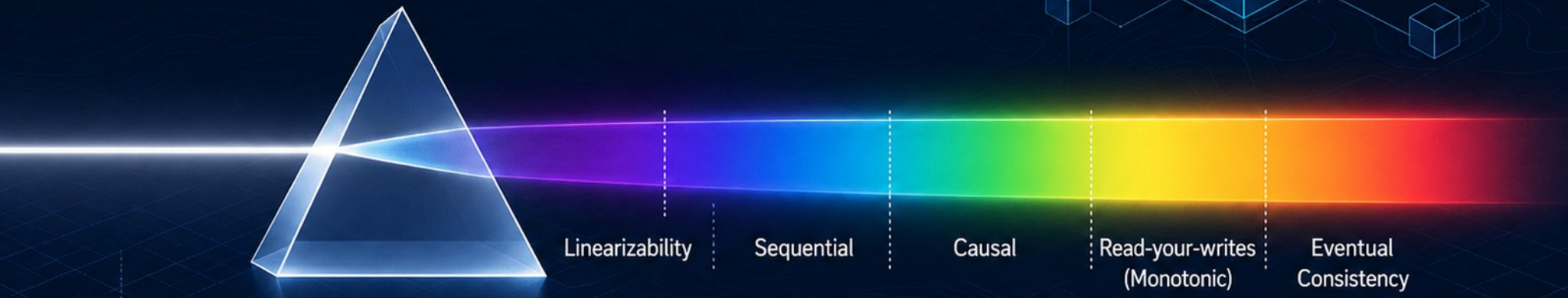
- **Synchronous vs. asynchronous** system models, plus partial synchrony and failure models (crash, omission, Byzantine).
- **Communication primitives:** message passing, RPC, and serialization (JSON, Protobuf, Avro).
- **Architectural paradigms:** client-server, peer-to-peer, and publish-subscribe, each suited to different needs.
- **The end-to-end argument:** why application-level guarantees often beat network-level ones (e.g., app-level retries vs. TCP).
- Middleware hides heterogeneity, presenting a distributed system as a single coherent entity.



The CAP Theorem and Consistency Trade-Offs

- During a partition, choose linearizability (C) or availability (A); partition tolerance is mandatory
- CAP applies only when the network is split; outside it, you can have both C and A
- Use CAP as a per-operation design prompt, not a global system label
- PACELC: Else (normal operation), trade off Latency (L) vs. Consistency (C)
- CP examples: ZooKeeper, etcd, Spanner. AP examples: Cassandra, DynamoDB, Couchbase

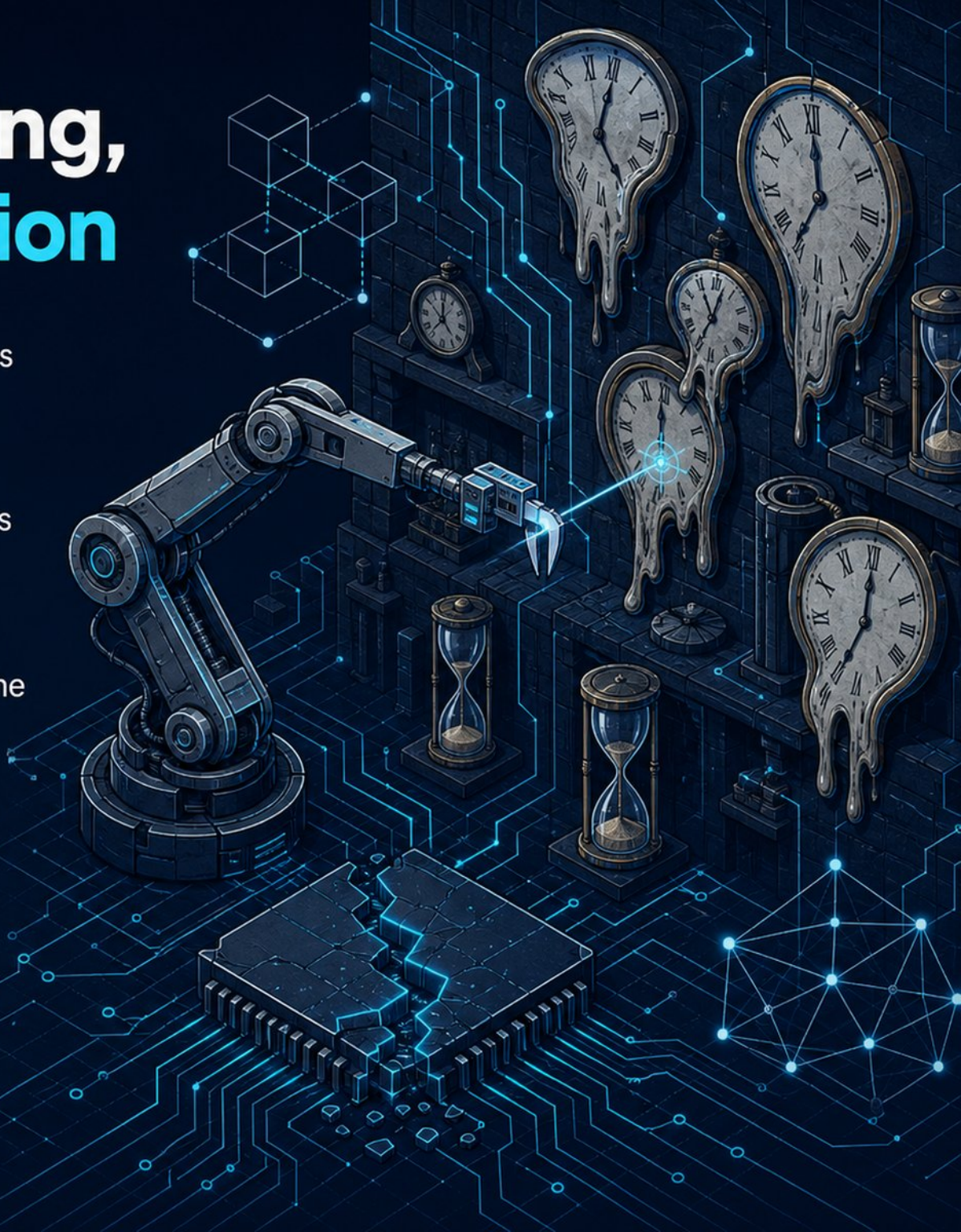
Consistency Models and Conflict Resolution



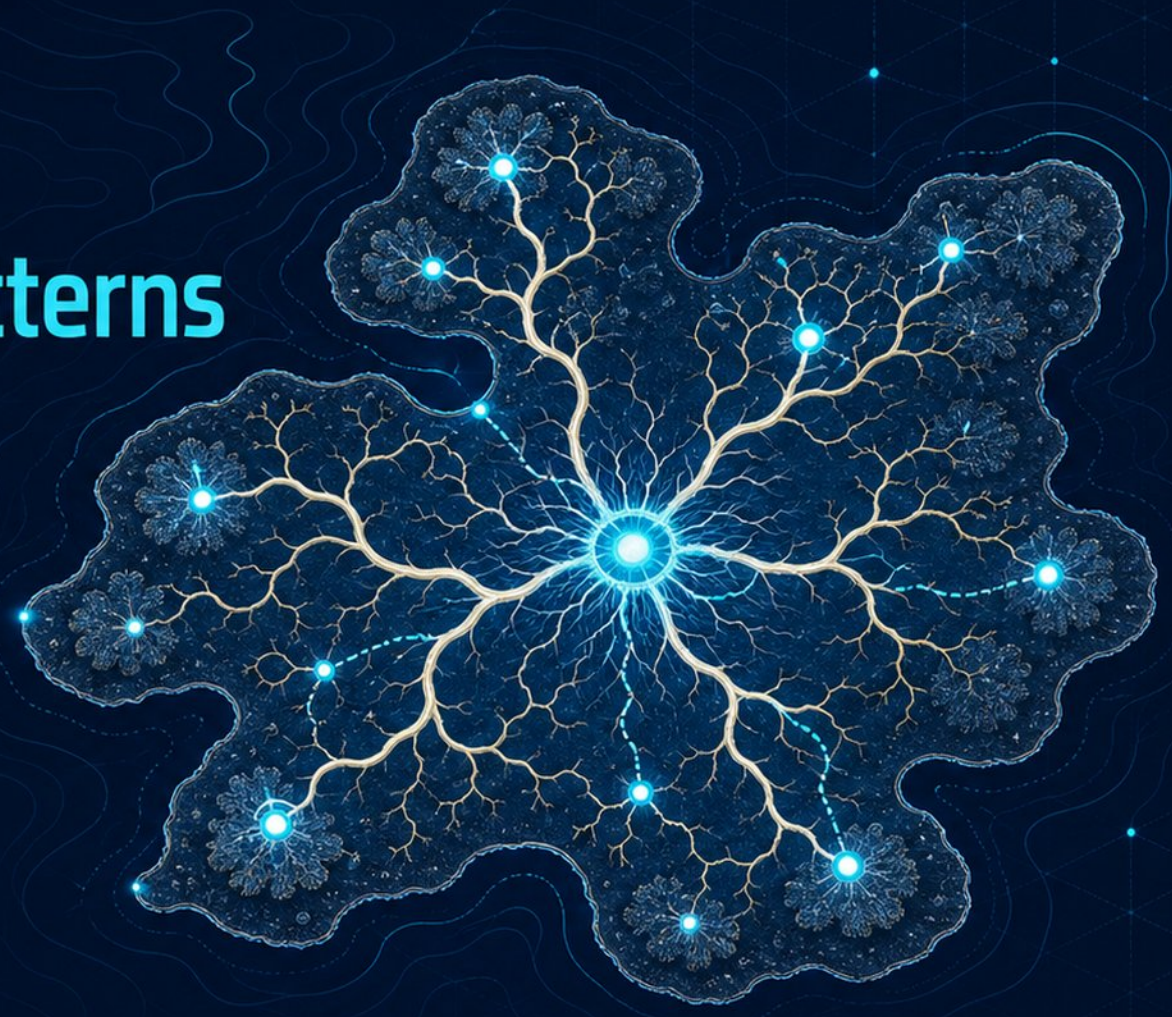
- Linearizability -> sequential -> causal -> eventual: a spectrum of guarantees, not a binary choice
- Read-your-writes and monotonic reads deliver practical consistency for user-facing apps
- CRDTs mathematically guarantee convergence without coordination (counters, sets, registers)
- Conflict resolution: Last-Write-Wins, custom merge functions, and vector-clock causality tracking
- Quorum math (R+W>N) ensures read freshness but does not alone provide linearizability

Timing, Ordering, and Coordination

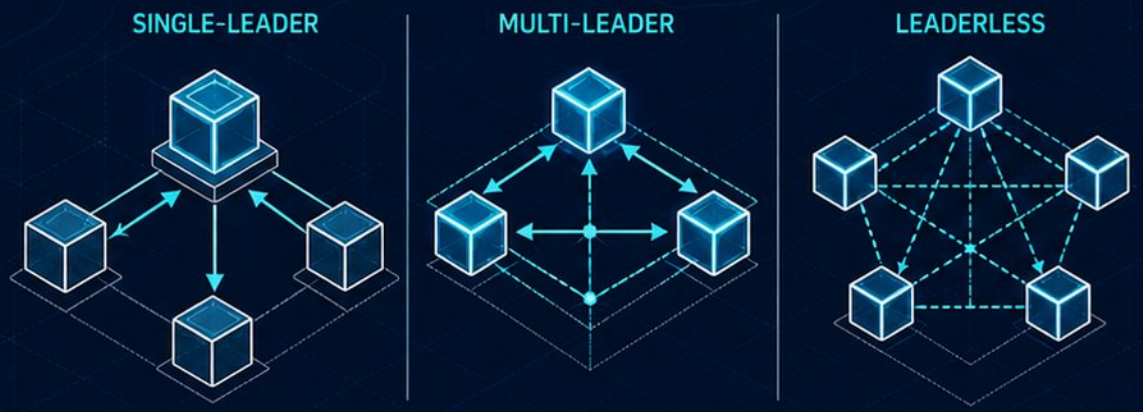
- No global clock — physical timestamps drift and are unreliable for ordering
- Logical clocks: Lamport timestamps track "happened-before," vector clocks capture causality
- Leader election and mutual exclusion are needed when nodes must act as one
- ZooKeeper and etcd provide locks, configuration, and group membership
- Leases and heartbeats detect failures without requiring perfect synchrony



Replication:
Strategies and Patterns



- Replication increases availability, fault tolerance, and read scalability
- Architectures: single-leader, multi-leader, and leaderless
- Synchronous replication ensures durability; asynchronous minimizes write latency
- Read-after-write consistency: route reads to the primary or use session guarantees
- Real-world: MySQL Group Replication, MongoDB replica sets, Redis, cloud-native



The Consensus Problem: Paxos and Raft

- **Consensus:** agreeing on a single value across nodes despite failures
- **Paxos:** proposers, accepters, learners — hard to implement correctly
- **Raft:** leader election, log replication, safety — designed for understandability
- **Multi-Paxos & reconfiguration:** practical performance, common pitfalls
- **QuePaxa (Meerkat):** leaderless consensus avoiding timeout tyranny at global scale
- **Consensus in production:** etcd/Raft (Kubernetes), CockroachDB, Spanner, Chubby

Distributed Transactions and Concurrency Control

- **ACID** is hard across services; each owns its own database
- **2PC** blocks under coordinator failure; rarely used in microservices
- **3PC** is non-blocking in theory but impractical in production
- **OCC** and **Percolator** handle large-scale transactions without 2PC
- **Sagas** use compensating local transactions for long workflows

The Saga Pattern in Practice

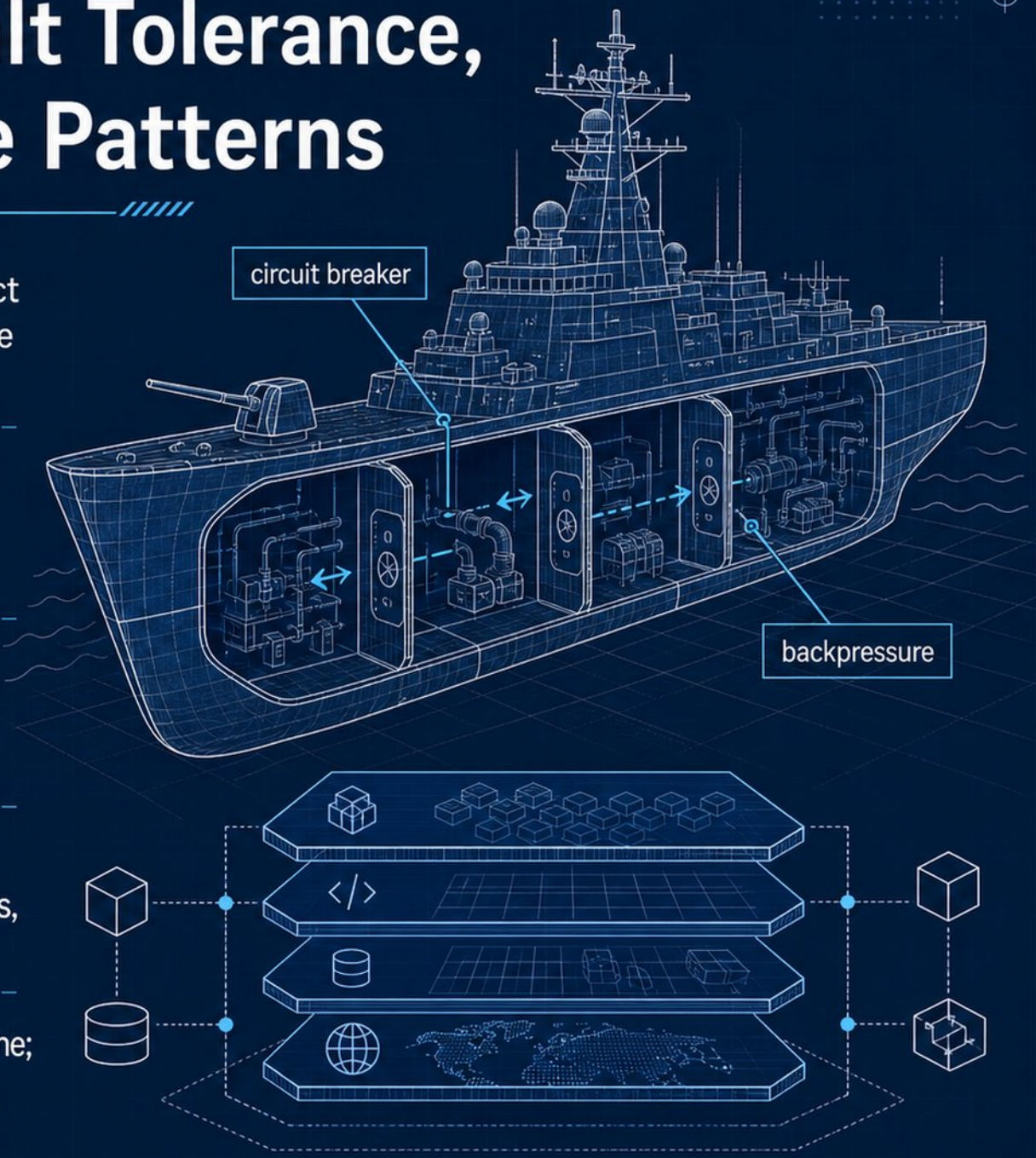
- Saga: local transaction chain with compensating undo, no global locks or XA.
- Choreography: peer events; Orchestration: central state machine.
- Steps: compensable, pivot (point of no return), retriable (idempotent).
- Compensations: semantic undo, must be idempotent and eventually succeed.
- Must-haves: outbox pattern, idempotency keys, semantic locks, saga state.

Scalability and Partitioning

- Vertical vs. horizontal scaling: Amdahl's Law and Universal Scalability Law
- Functional decomposition (microservices) combined with data partitioning
- Sharding strategies: key-range, hash-based, and directory-based
- Consistent hashing minimizes data movement during topology changes
- Hot-spot mitigation via key salting, dynamic rebalancing, and caching patterns

Reliability, Fault Tolerance, and Resilience Patterns

- Perfect vs. eventually perfect failure detectors; impossible in pure async systems
- Prevent cascading failures: circuit breakers, bulkheads, retries with backoff
- Idempotency keys and database constraints for exactly-once semantics
- Load balancing strategies: round-robin, least connections, consistent hashing
- Chaos engineering as a discipline; Litmus, Chaos Mesh, Gremlin, Krkn, Entropy, Tumult



Chaos Engineering in the 2026 Ecosystem

- Systematic resilience testing uncovers weaknesses before incidents do
- CNCF projects (LitmusChaos, Chaos Mesh) and enterprise platforms (Gremlin, Steadybit, Harness)
- Developer-friendly tools: Entropy (agentless, ephemeral containers), Tumult (Rust, native OTel, DuckDB), Pastaay, Krkn
- AI-assisted chaos: autonomous experiment generation, SLO-aware fault injection, fitness-based scenario evolution
- From GameDays to CI/CD: chaos experiments integrated into pipelines as standard practice

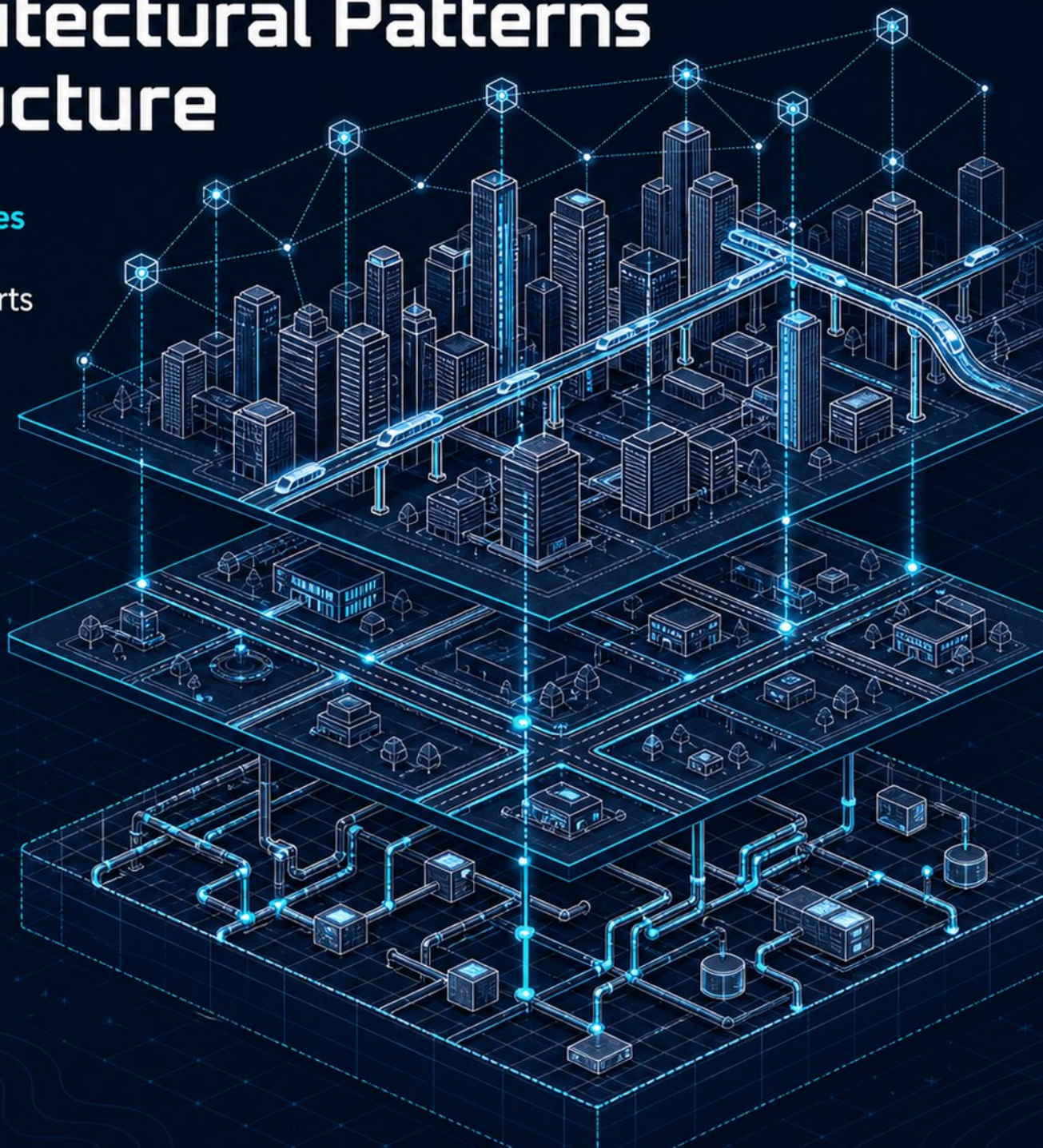
OBSERVABILITY & DEBUGGING

IN DISTRIBUTED DEPLOYMENTS

- Traces reveal “why” across services; metrics only show “what”
- W3C Trace Context propagates context; breaks fragment traces
- Tail-based sampling keeps 100% of errors and slow traces
- Inject trace_id/span_id into logs for full-context debugging
- RED metrics (Rate/Errors/Duration) drive proactive alerting

Modern Architectural Patterns and Infrastructure

- **Edge functions + edge databases** (D1, Turso, Neon) redefine deployment with sub-ms cold starts
- **WebAssembly** (WASI 0.3, Component Model) enables polyglot, sandboxed edge microservices
- **Service meshes** (Istio, Linkerd) offload retries, circuit breaking, and observability
- **Event-driven patterns:** event sourcing, CQRS, and durable execution platforms like Temporal
- **AI-driven orchestration:** NVIDIA AI Grid, Google Agent Executor, and federated Kubernetes (k0smos, CODECO)



Key Takeaways and Further Learning

- Distributed systems are defined by concurrency, lack of a global clock, and independent failures
- For each operation, ask: what failure model, which consistency, and what happens during a partition?
- Essential toolbox: consensus, replication, sharding, sagas, tracing, and chaos engineering
- Foundational papers (Dynamo, Raft, Spanner, etc.) provide lasting mental models for modern systems
- Explore CNCF, Papers We Love, MIT 6.5840, and the open-source distributed systems community

PersonWise.

PersonWise • AI digital-human course platform

Scan the code or click anywhere to watch this course live, with voice Q&A

personwise.ai/t/fbb9eb42/public