# Nexus Blueprint 2.0: A State-of-the-Art, Agent-Governed Event-Sourced Microservice Foundation

## I. Strategic Imperative and Compliance Framework

### 1.1. Executive Summary: Maximizing Value Through Architectural Foundation

The Nexus Blueprint 2.0 represents an expert-level implementation of the Event Sourcing (ES) and Command Query Responsibility Segregation (CQRS) patterns, specifically engineered to fulfill the stringent requirements of the Kiroween Hackathon's Skeleton Crew category.1 The architecture is designed for maximum scoring potential, achieving high marks in Implementation through advanced Kiro utilization and high Potential Value through enterprise-grade resilience and auditability.1 The core thesis is that a serverless, event-sourced microservice foundation, leveraging the append-only nature of the event store, offers resilience, auditability, and massive scalability unattainable by traditional CRUD models.2

Strategically, the Blueprint is positioned not merely as a competition entry but as a foundational Software as a Service (SaaS) boilerplate.1 This strategic framing allows the project to target the high-value prize stacking combination: winning the Best Skeleton Crew Category Prize ($5,000) and concurrently qualifying for the Best Startup Project Bonus Prize ($10,000).1 To secure this dual award, the architecture must demonstrate immediate commercial viability, which is achieved through its inherent scalability, low operational cost via serverless compute (AWS Lambda), and robust architectural governance enforced by the Kiro agentic toolchain.4

### 1.2. Deconstructing the Skeleton Crew Mandate (Repo A vs. Repo B)

The Skeleton Crew category imposes a specific structural constraint: the submission must demonstrate the foundation's versatility by supporting two distinct applications derived from the single blueprint, materialized physically as two separate repository folders.1 The Nexus Blueprint 2.0 leverages the fundamental separation of concerns inherent in the CQRS pattern to natively satisfy this requirement in an architecturally sound manner.1

The separation between the transactional Write Model (Command Service) and the reporting Read Model (Query Dashboard) provides the necessary objective distinction required for compliance. The resulting components are segregated as follows:

1. **Repository A (Command Service):** This houses the Write Model. Its function is to handle all transactional user commands, execute business logic validation, and publish events to the core Event Store.1 This service is characterized by its high consistency demands and relatively low instance count, focused on minimizing merge conflicts.7
2. **Repository B (Query Dashboard):** This houses the Read Model. Its function is to consume events asynchronously and maintain highly optimized, denormalized data projections tailored specifically for reporting, analytics, and user interface display.1 This service is characterized by high read throughput and is designed for horizontal scaling to handle large query volumes.7

This architectural choice elevates the project’s Potential Value score because the distinction between the two applications is fundamental, rooted in distinct data models, access patterns, and scaling needs, rather than superficial functional segregation. This mandates an architecture that is inherently more resilient and scalable than a system attempting to handle both high-volume transactional writes and complex analytical reads within a single database paradigm.2

Table: Nexus Blueprint Component Segregation (Skeleton Crew Compliance)

| **Component Category** | **Role/Function** | **Repository Folder** | **Kiro Artifact Used** |
| --- | --- | --- | --- |
| Event Store (Foundation) | Single Source of Truth, Immutability, Audit Trail | Shared Configuration/Infrastructure | Steering Docs (structure.md) |
| Command Service (Write Model) | Handles Transactional Commands (C), Validation, Event Publication | Repository A (App 1) | Spec-Driven Generation |
| Query Dashboard (Read Model) | Handles Optimized Queries (Q), Projections, Analytics | Repository B (App 2) | Agent Hooks (Consistency enforcement) |
| Event Broker/Bus | Decoupled Communication Backbone (Async Transport) | Shared Configuration/Infrastructure | Steering Docs (Protocols) |

### 1.3. The Kiro Implementation Strategy for Maximum Score (The Governance Contract)

Achieving a top score in the Implementation criterion necessitates demonstrating a deep, strategic understanding of Kiro features beyond basic code generation.1 In the context of a complex ES/CQRS foundation, Kiro is deployed as the **Architectural Sentinel**, enforcing architectural consistency across the asynchronous boundary separating Repository A and Repository B.8

The architectural challenge in decoupled systems like ES/CQRS is preventing "architectural drift," where developers introduce coupling or anti-patterns out of convenience.9 Kiro’s tools address this directly:

* **Specs** define the contract: Specifications formalize the single, shared data contract (Commands and Events), ensuring the write side (Repo A) and the read side (Repo B) are working with the identical, version-controlled domain artifact.1
* **Steering Docs** define the law: These persistent constraints enforce non-negotiable architectural principles, such as prohibiting synchronous Remote Procedure Calls (RPC) between the two services, ensuring communication remains strictly asynchronous via the Event Broker.1
* **Hooks** perform real-time governance: Event-driven automations trigger compliance checks during the development workflow, such as instantly validating that new domain event schemas adhere to the mandatory ES protocol metadata (e.g., including aggregateID and timestamp).1

By encoding these architectural laws into the Kiro toolchain, the Blueprint automates the prevention of architectural erosion, guaranteeing that the inherent resilience and scalability of the ES/CQRS structure are maintained throughout development. This strategic use of Kiro transforms it from a coding assistant into an essential governance layer, which directly maximizes the Implementation score.1

## II. Reference Architecture: Nexus Blueprint 2.0 (Serverless ES/CQRS)

### 2.1. The Event Sourcing Core (The Immutable Skeleton)

Event Sourcing is adopted because it replaces traditional mutable state persistence (CRUD) with an immutable, append-only log of domain events, providing a perfect historical audit trail.2 This immutability is the "skeleton" shared across both Repo A and Repo B. The Event Store serves as the single source of truth, from which all current system state can be reliably reconstructed by replaying the sequence of events.3

For the Nexus Blueprint 2.0, the Event Store implementation leverages Amazon DynamoDB. DynamoDB is highly suitable for this role due to its high write throughput, essential for handling high-volume transactional events, and its native integration with DynamoDB Streams for change data capture (CDC).4 The persistence layer guarantees fast writes because it only performs atomic append operations, inherently avoiding the database locking and resource contention issues that plague high-load CRUD systems.2

### 2.2. Serverless Component Selection: The AWS Foundational Stack

To support the high-scale, low-operational-cost mandate required for the Best Startup Project category, the architecture is entirely serverless.1 This minimizes infrastructure management overhead and aligns the cost structure with actual usage.5

* **API Gateway:** Amazon API Gateway provides a managed, scalable entry point, essential for reliably routing incoming requests. It is configured to route write-side operations (commands, e.g., POST/PUT) exclusively to the Command Service Lambda functions (Repo A), and read-side operations (queries, e.g., GET) to the Query Dashboard Lambda functions (Repo B).4 This separation of C and Q endpoints reinforces the CQRS boundary.
* **Compute:** AWS Lambda functions are the primary compute resource for both the Command Handlers in Repo A and the Projection Handlers in Repo B.4 Lambda is event-driven, scaling instantly to meet demand without requiring provisioned capacity, thus ensuring elasticity for a foundational architecture.5
* **Read Persistence:** The Query Dashboard (Repo B) employs polyglot persistence, utilizing specialized data stores like Amazon Aurora Serverless and Amazon OpenSearch to optimize query performance, a key benefit of the CQRS pattern.4

### 2.3. Event Flow and Propagation via Managed Streaming Services

Effective event propagation is critical to maintaining eventual consistency between the Write Model (Repo A) and the Read Model (Repo B).3 The Nexus Blueprint 2.0 implements a robust, resilient strategy known as Topic Queue Chaining using managed AWS services.4

The event flow proceeds through three layers:

1. **Event Capture:** Events are captured by DynamoDB Streams immediately upon being appended to the Event Store.4
2. **Stream Processing and Fan-Out:** A dedicated AWS Lambda function processes the raw stream data, validates it, and publishes the domain event to a central **Amazon SNS Topic**.4 This SNS Topic acts as the central event bus for the architecture.
3. **Projection Isolation:** Each unique Read Model (projection) subscribes to the SNS Topic via its own dedicated **Amazon SQS Queue**.4

This layered approach significantly increases resilience. If an individual Projection Handler (in Repo B) fails due to internal errors or downstream database issues, the corresponding event message remains safely preserved in its dedicated SQS queue.4 This isolation ensures that the failure of one projection does not halt the entire system or compromise the ability of other projections to update their states, demonstrating superior architectural robustness and potential for scale.7

## III. Application 1: The Command Service (Repo A - Write Model)

### 3.1. Domain-Driven Design (DDD) Aggregates and Consistency Boundary

The Command Service (Repo A) is implemented using Domain-Driven Design (DDD) Aggregates.15 An aggregate, such as Order or BankAccount, acts as a consistency boundary, ensuring that all related objects are always in a consistent state before any new event is appended.7 This consistency is crucial because the events published by Repo A are the immutable source of truth for the entire system.6

When a user issues a command (e.g., IssueTransferCommand), the Command Handler (a Lambda function in Repo A) must first validate the action. This involves rehydrating the current state of the relevant aggregate instance (e.g., BankAccount-1234) by reading and replaying all past events for that specific instance from the Event Store.15 Only after the handler confirms the action is valid against the recreated state is a new Domain Event (e.g., TransferInitiatedEvent) created and appended to the Event Store.15 Kiro’s Specification integration ensures that all generated command handlers adhere to these established DDD boundaries and event versioning requirements.16

### 3.2. Performance Optimization: Strategic Snapshotting Layer Implementation

A critical challenge in classic ES is the performance degradation that occurs when an aggregate accumulates thousands or millions of events.15 Replaying a very long event stream to reconstitute state before processing a new command introduces latency, compromising the Command Service’s performance.17

The Nexus Blueprint 2.0 mitigates this with a strategic Snapshotting Layer.17 A dedicated, separate DynamoDB table is used to store calculated snapshots of aggregate state at predefined intervals. When a command is received, the system loads the most recent snapshot and replays only the few events that occurred *after* that snapshot, drastically reducing the time required for state reconstruction and improving Command Service responsiveness.15

To maintain this performance optimization, a Kiro Agent Hook is deployed for automated snapshot management.1 This hook is configured to monitor the event stream length metric. If an aggregate stream exceeds a threshold (e.g., 5,000 events), the hook automatically prompts Kiro to generate or update the background job (e.g., a scheduled Lambda function) responsible for calculating and storing the next snapshot.1 This automation ensures continuous performance efficiency and demonstrates advanced use of Kiro hooks for system maintenance.2

### 3.3. Write Model Persistence and Atomic Transaction Guarantees

The DynamoDB Event Store, implemented using single-table design principles, guarantees fast, atomic writes for the single operation of appending a new event.6 This capability eliminates the need for expensive distributed transactions that typically span both a database and a message broker, significantly improving write performance.6

In compliance with Domain-Driven Design and ES best practices, the Event Store employs horizontal partitioning based strictly on the Aggregate ID.19 While traditional relational sharding often uses complex business keys (like geographical location or user ID), the literature confirms that for an Event Store, the Aggregate ID is the only necessary partition point.19 Adhering to Aggregate ID sharding simplifies complexity and allows for clean aggregate mobility and streamlined horizontal scaling.19 Introducing additional logical keys for sharding (e.g., embedding location keys into the ID) is explicitly avoided as it complicates aggregate lifecycle management and consistency, reinforcing the commitment to architectural purity and high performance.2

## IV. Application 2: The Query Dashboard (Repo B - Read Model)

### 4.1. Advanced Query Strategy: Polyglot Persistence Implementation

The central benefit of CQRS is the ability to decouple the read workload from the write workload, allowing the use of specialized data models for querying.14 Repository B utilizes an advanced strategy of Polyglot Persistence, selecting the best data store technology for each specific query pattern, thereby maximizing read performance and extensibility.20

The Nexus Blueprint 2.0 incorporates a three-store strategy for the Read Model:

1. **High-Volume Structured Queries:** Amazon Aurora Serverless (Relational Database Service) is used for common structured list views, filtering, and standard reporting. This provides the familiarity and structural integrity of an RDBMS for complex joins and predictable queries.4
2. **Search and Analytics:** Amazon OpenSearch is employed for fuzzy, full-text searches, heavy analytical aggregations, and high-dimensional querying. Utilizing a specialized search index drastically improves performance for non-exact match queries compared to relational stores.14
3. **Temporal Queries (Historical):** Direct access to the DynamoDB Event Store (read-only access) is retained for specialized temporal queries, such as audit or "time travel" requests.21

The use of Kiro Vibe Coding is critical within Repo B to rapidly generate the specific data transformation scripts (projection logic) required to convert the standardized domain events into the highly optimized, denormalized schemas of these polyglot data stores.1 This automation capability addresses the increased complexity inherent in managing multiple data technologies.

### 4.2. Read Model Projections and Event Processor Architecture

Projection Handlers are the Lambda functions housed in Repo B that consume events and update the Read Model data stores.13 This layer is designed for maximum resilience and isolation. Each specific polyglot store (e.g., Aurora, OpenSearch) is updated by its own dedicated Projection Handler Lambda, which polls its dedicated SQS queue.4

This architectural isolation ensures that if the event ingestion for one specific projection fails, the message is retained safely in its isolated SQS queue, and other independent projections continue to function without interruption.7 The Query Dashboard’s ability to quickly and reliably update multiple, optimized data views based on a single source of truth (the Event Store) demonstrates significant architectural polish and high Quality and Design scores.

### 4.3. Managing Eventual Consistency and Read-After-Write Concerns

The fundamental trade-off of using a decoupled, asynchronous architecture like CQRS is accepting eventual consistency.3 This means that the Read Model (Repo B) will lag the Write Model (Repo A) by some milliseconds or seconds, creating a liveness guarantee rather than an immediate safety guarantee.22 This scenario, while necessary for scalability, introduces complexity for developers accustomed to ACID transactions and can lead to subtle bugs.22

The Nexus Blueprint 2.0 addresses this complexity with two primary mitigations:

1. **Event Replay for Auditing:** The inherent auditability of Event Sourcing is the key resilience factor, allowing for temporal reconstruction of state.2 The design explicitly facilitates the creation of a "staging" environment where developers can trigger event replays to audit their projection logic and verify that eventual consistency convergence occurs reliably, directly addressing the difficulty of debugging eventual consistency in production environments.23
2. **Optimistic Consistency for UX:** For critical user interactions requiring immediate feedback (the "read-after-write" pattern), the Command Service (Repo A) returns an aggregate version identifier or event ID upon successful command execution. The Query Dashboard (Repo B) can use this identifier to poll its projection until the required state or event version is observed, providing the end user with the illusion of immediate consistency while maintaining the high scalability of eventual consistency.24

## V. Kiro Agentic Governance: The Blueprint Automation Layer

### 5.1. Spec-Driven Development: Generating the Dual-Application Scaffolding

Spec-Driven Development replaces ad hoc, prompt-driven coding with a structured approach where a unified specification acts as a durable, version-controlled "super prompt" for the AI agents.10 In the Nexus Blueprint 2.0, Kiro Specs are instrumental in maintaining architectural homogeneity across the two required repository folders (Repo A and Repo B).1

The Kiro Specs define the central domain structure, including precise data models for all Commands and Events.25 Kiro then uses this single specification to simultaneously generate the repetitive boilerplate code in both repositories 1: command handlers and DTOs in Repo A, and event listeners and projection stubs in Repo B. This capability guarantees that the core domain contract remains perfectly synchronized, eliminating the coordination overhead and integration bugs typically found between teams working on decoupled services.8

### 5.2. Steering Docs: Enforcing the Asynchronous Protocol and Domain Boundaries

Steering Documents provide Kiro with persistent architectural knowledge, functioning as the architectural constitution that dictates how code must be generated.3 For a complex pattern like ES/CQRS, the Steering Docs enforce critical laws that prevent architectural anti-patterns.9

The structure.md Steering Document contains explicit mandates to enforce the integrity of the decoupled microservices:

* **Decoupling Enforcement:** The document explicitly prohibits Kiro from generating any code in Repo A that performs direct synchronous calls to endpoints or services residing in Repo B, and vice versa. Kiro is mandated to only generate code that publishes events to the Event Broker (SNS/SQS), ensuring the system remains highly decoupled and resilient.9
* **Immutability Rule:** Steering Docs enforce that the Write Model (Repo A) must only generate code that appends to the Event Store, strictly prohibiting destructive update or delete operations on the event stream itself.2

By operationalizing these architectural principles via Steering Docs, the Blueprint ensures that Kiro acts as a persistent governance mechanism, automatically correcting or refusing to generate code that would violate the fundamental constraints of the ES/CQRS pattern, ensuring the long-term scalability of the foundation.9

### 5.3. Agent Hooks: Automated Compliance Checks and Consistency Maintenance

Agent Hooks are event-driven automations that trigger specific AI actions based on developer activity, such as saving or creating a file.11 These hooks are configured to automate protocol enforcement and maintain consistency between the decoupled services (Repo A and Repo B).11

Two critical Agent Hooks are implemented:

1. **Event Schema Validation Hook:** This hook is configured to trigger whenever a new Domain Event definition file is created or saved within Repo A’s source code. The hook immediately validates that the new schema adheres to the mandatory ES protocol, checking for the existence of required metadata fields such as eventID, timestamp, and aggregateID.1 This proactive validation ensures that the events consumed by Repo B are guaranteed to contain all necessary data for processing and auditing, safeguarding data integrity at the source.
2. **Projection Synchronization Hook:** This hook triggers upon a change (e.g., modifying or adding a field) in an existing Domain Event schema within Repo A. Its action is to automatically generate an associated update task or alert within Repo B’s tasks list or documentation. This automated prompt forces the projection development team to immediately address the required changes for event schema evolution (e.g., updating their upcasting logic or read model transformations).16 This synchronization reduces manual coordination overhead and minimizes the risk of deserialization errors due to schema drift.

## VI. Next-Generation Capabilities and Commercialization

### 6.1. Handling Event Schema Evolution (Upcasting and Side-by-Side Strategy)

A state-of-the-art ES foundation must include a robust strategy for handling event schema evolution, as events, once written, are immutable.2 The Blueprint mandates a dual-strategy approach to ensure long-term compatibility 28:

1. **Backward Compatibility Enforcement:** Kiro Steering Docs enforce strict, additive-only changes for event schemas. Developers are forbidden from renaming or removing existing fields; new fields must always be optional and include default values.29 This ensures that older consumers (in Repo B) can safely ignore new fields without failure.
2. **Upcasting Transformation:** For structural changes that cannot be additive (e.g., splitting a single field into two), the Projection Handlers (Repo B) implement an Upcasting mechanism.27 This logic transforms older event versions into the current schema format on-the-fly as they are read from the event stream, ensuring that historical data remains usable by modern application logic.2 Furthermore, integration with a Schema Registry centralizes governance, validating schema changes and compatibility checks before events are published, mitigating the risk of data incompatibility between the producer (Repo A) and consumers (Repo B).29

### 6.2. Temporal Queries and "Time Travel" Feature Design

The Nexus Blueprint 2.0 leverages the immutability of the event log to offer "time travel" functionality—a high-value feature often required for detailed audits, regulatory compliance, and retrospective business analysis.2

This capability is exposed via a dedicated Temporal Query API endpoint within the Query Dashboard (Repo B). The associated Lambda function is designed to bypass the standard, denormalized read models and instead reads the raw event stream directly from the DynamoDB Event Store. The function executes the query by applying a time-based filter (such as the upperBound parameter) to limit the event replay to a specific historical timestamp.21 By rehydrating the relevant aggregate state up to that exact moment, the system can answer queries like, "What was the system's inventory level 60 days ago?".21 To optimize performance for these queries, Kiro Specs mandate that all relevant timestamp fields are correctly structured (e.g., using UTC formatting) to ensure they are SARGable (search argumentable) for efficient filtering.30

### 6.3. Nexus Blueprint as a SaaS Boilerplate: Path to Startup Prize and Commercial Viability

The evolved Nexus Blueprint is strategically positioned as the core product of an architecture startup, designed to capture the Best Startup Project Bonus Prize.1 The commercial value proposition rests on mitigating the known enterprise pain points associated with adopting complex distributed patterns like ES/CQRS.2

The foundation's competitive advantage is threefold:

1. **Complexity Abstraction:** The serverless architecture and robust event propagation patterns (Topic Queue Chaining) abstract away the manual complexity and operational overhead required to set up a resilient ES/CQRS system from scratch.4
2. **Built-in Architectural Governance:** The integration of Kiro’s Agentic Governance (Specs, Steering Docs, and Hooks) is the key feature that guarantees architectural integrity.8 This governance layer ensures that as the codebase and development team scale, the complex protocol requirements of ES/CQRS are automatically enforced, preventing the chaotic development and inconsistent implementations common in large agent-assisted projects.8
3. **Auditability and Resilience:** The inherent support for time travel and the Topic Queue Chaining pattern ensures the solution is immediately resilient and compliant with high-stakes auditing requirements, making it attractive for finance, logistics, or healthcare sectors.2

By providing an inherently scalable, governed, and resilient foundation that is ready for immediate commercial development, the Nexus Blueprint 2.0 demonstrates superior Potential Value and provides the necessary credibility to secure the Best Startup Project Bonus Prize.1

## VII. Conclusion and Recommendations

The Nexus Blueprint 2.0 provides a state-of-the-art serverless implementation of the Event Sourcing and CQRS patterns, strategically engineered for maximum performance and architectural resilience. The system successfully meets the Skeleton Crew mandate by naturally separating the Write Model (Repo A) and the Read Model (Repo B) based on their fundamental responsibilities and distinct data persistence needs (Polyglot Persistence).

The central technical innovation lies in deploying Kiro as the Architectural Sentinel. Through Kiro Specs, Steering Docs, and Agent Hooks, the architecture automates the governance of complex ES/CQRS protocols, ensuring consistent event schema validation, enforcing architectural decoupling, and automating critical performance tasks like snapshot management. This advanced utilization of Kiro directly ensures a top score in the Implementation criterion and provides a unique selling point for commercialization.

The recommendation is to proceed immediately with development, focusing on formalizing the core Event and Command DTOs within the Kiro Specs, as these artifacts serve as the single source of truth for scaffolding both Repository A and Repository B. Subsequent efforts should prioritize configuring the Steering Docs to strictly enforce the asynchronous communication mandate, guaranteeing that the resilience designed into the Topic Queue Chaining propagation model is preserved throughout development.

#### Works cited

1. Kiroween\_hackathon.docx
2. Event Sourcing pattern - Azure Architecture Center | Microsoft Learn, accessed November 1, 2025, <https://learn.microsoft.com/en-us/azure/architecture/patterns/event-sourcing>
3. Event Sourcing vs Event-Driven Architecture: Core Contrasts | Estuary, accessed November 1, 2025, <https://estuary.dev/blog/event-driven-vs-event-sourcing/>
4. Scaling Distributed Systems with Event Sourcing, CQRS, and AWS Serverless, accessed November 1, 2025, <https://builder.aws.com/content/2dqfVdKPEO1Q917YG25q8F3Y2QW/scaling-distributed-systems-with-event-sourcing-cqrs-and-aws-serverless>
5. Revolutionizing Your Architecture: A Beginner's Guide to Serverless CQRS and Event Sourcing | by Ahmed Mohiuddin | Medium, accessed November 1, 2025, <https://medium.com/@ahmed.mohiuddin.architecture/revolutionizing-your-architecture-a-beginners-guide-to-serverless-cqrs-and-event-sourcing-40a412c58dc3>
6. Build a CQRS event store with Amazon DynamoDB | AWS Database Blog, accessed November 1, 2025, <https://aws.amazon.com/blogs/database/build-a-cqrs-event-store-with-amazon-dynamodb/>
7. CQRS Pattern - Azure Architecture Center | Microsoft Learn, accessed November 1, 2025, <https://learn.microsoft.com/en-us/azure/architecture/patterns/cqrs>
8. Spec-Driven Development & AI Agents Explained - Augment Code, accessed November 1, 2025, <https://www.augmentcode.com/guides/spec-driven-development-ai-agents-explained>
9. Using AI Agents to Enforce Architectural Standards | by Dave Patten - Medium, accessed November 1, 2025, <https://medium.com/@dave-patten/using-ai-agents-to-enforce-architectural-standards-41d58af235a0>
10. Spec-Driven Development: The Key to Scalable AI Agents - The New Stack, accessed November 1, 2025, <https://thenewstack.io/spec-driven-development-the-key-to-scalable-ai-agents/>
11. What Is Kiro? Agentic AI IDE Definition & Workflow Explained - Skywork.ai, accessed November 1, 2025, <https://skywork.ai/blog/kiro-agentic-ai-ide/>
12. Event Sourcing vs Event Driven Architecture difference - Stack Overflow, accessed November 1, 2025, <https://stackoverflow.com/questions/71083541/event-sourcing-vs-event-driven-architecture-difference>
13. Decompose monoliths into microservices by using CQRS and event sourcing - AWS Prescriptive Guidance, accessed November 1, 2025, <https://docs.aws.amazon.com/prescriptive-guidance/latest/patterns/decompose-monoliths-into-microservices-by-using-cqrs-and-event-sourcing.html>
14. From SQL Headaches to Scalable Design: A Guide to the CQRS Pattern - Dev Genius, accessed November 1, 2025, <https://blog.devgenius.io/from-sql-headaches-to-scalable-design-a-guide-to-the-cqrs-pattern-2dbfd27c4a01>
15. CQRS + Event Sourcing for the Rest of Us : r/softwarearchitecture - Reddit, accessed November 1, 2025, <https://www.reddit.com/r/softwarearchitecture/comments/1l0yobs/cqrs_event_sourcing_for_the_rest_of_us/>
16. Best practices - Docs - Kiro, accessed November 1, 2025, <https://kiro.dev/docs/specs/best-practices/>
17. The Performance Factor in Event Sourcing: What You Need to Know - patchlevel, accessed November 1, 2025, <https://patchlevel.de/blog/the-performance-factor-in-event-sourcing>
18. Introducing Kiro, accessed November 1, 2025, <https://kiro.dev/blog/introducing-kiro/>
19. Event Sourcing and Horizontal Partitioning/Sharding - Google Groups, accessed November 1, 2025, <https://groups.google.com/g/dddcqrs/c/LXcfPrAoDcs>
20. Polyglot Persistence: A Strategic Approach to Modern Data Architecture - Medium, accessed November 1, 2025, <https://medium.com/@rachoork/polyglot-persistence-a-strategic-approach-to-modern-data-architecture-e2a4f957f50b>
21. Patterns for Temporal Queries - EventSourcingDB, accessed November 1, 2025, <https://docs.eventsourcingdb.io/best-practices/patterns-for-temporal-queries/>
22. Eventual consistency - Wikipedia, accessed November 1, 2025, <https://en.wikipedia.org/wiki/Eventual_consistency>
23. An Empirical Study on Challenges of Event Management in Microservice Architectures, accessed November 1, 2025, <https://arxiv.org/html/2408.00440v1>
24. Microservices, CQRS: Eventual consistency vs Strong consistency(Read after Write consistency) - Stack Overflow, accessed November 1, 2025, <https://stackoverflow.com/questions/48294450/microservices-cqrs-eventual-consistency-vs-strong-consistencyread-after-write>
25. Specs - Docs - Kiro, accessed November 1, 2025, <https://kiro.dev/docs/specs/>
26. What Is Spec-Driven Development? Tools, Process, and the Outcomes You Need To Know, accessed November 1, 2025, <https://www.epam.com/insights/ai/blogs/inside-spec-driven-development-what-githubspec-kit-makes-possible-for-ai-engineering>
27. Event Sourcing Explained: The Pros, Cons & Strategic Use Cases for Modern Architects, accessed November 1, 2025, <https://www.baytechconsulting.com/blog/event-sourcing-explained-2025>
28. Event Schema Evolution for API Gateways - DreamFactory Blog, accessed November 1, 2025, <https://blog.dreamfactory.com/event-schema-evolution-for-api-gateways>
29. Best Practices for Evolving Schemas in Schema Registry - Solace Docs, accessed November 1, 2025, <https://docs.solace.com/Schema-Registry/schema-registry-best-practices.htm>
30. Temporal Table Usage Scenarios - SQL Server - Microsoft Learn, accessed November 1, 2025, <https://learn.microsoft.com/en-us/sql/relational-databases/tables/temporal-table-usage-scenarios?view=sql-server-ver17>