YOUR AI IS ONLY AS GOOD AS YOUR SPECIFICATIONS

The Event-Driven Advantage

A technical whitepaper on AI-assisted development in event-driven architectures

AXONIQ

Allard Buijze, Axoniq | Martin Dilger, Nebulit | Adam Dymitruk, Adaptech

Executive Summary

TL;DR

AI coding assistants represent a fundamental shift in software development velocity. However, early adoption patterns reveal a critical oversight: these tools function as multipliers of existing development processes. Well-specified systems experience genuine productivity gains. Poorly specified systems accumulate technical debt at unprecedented rates.

This whitepaper examines the relationship between AI-assisted development and specification quality. It draws on observations from enterprise environments using Axon Framework. We demonstrate that event modeling combined with event sourcing provides a specification framework well-suited for AI-assisted development.

Key findings:

- AI coding assistants amplify whatever development process they encounter, including problematic ones
- Specification quality directly determines whether AI assistance accelerates value delivery or technical debt accumulation
- Event modeling provides unambiguous specifications that bridge business intent and technical implementation
- Pattern repetition in event-driven architectures gives AI clear, consistent examples to follow when generating code
- Event sourcing's complete historical context enables more effective AI reasoning about system behavior

Recommendations:

For organizations adopting AI-assisted development, we recommend specification-first approaches that provide clear boundaries and repeatable patterns. Event modeling with event sourcing represents one proven implementation of this principle, particularly for systems where long-term maintainability justifies upfront design investment.

1. Introduction

1.1 The AI Development Landscape

The software industry has rapidly adopted AI coding assistants. Tools like Claude Code and GitHub Copilot now assist millions of developers daily. These systems promise substantial productivity gains through automated code generation, intelligent completions, and rapid prototyping capabilities.

Recent observations, however, reveal complexity beneath the surface. Early adopters report mixed results: some teams experience genuine productivity gains, while others find these tools actually reduce productivity. This paradox demands investigation: how can tools designed to accelerate development sometimes produce the opposite effect?

1.2 The Missing Variable: Specification Quality

The answer lies not in the AI tools themselves. It lies in how development teams apply them.

AI coding assistants operate as force multipliers. They accelerate whatever process they encounter. When that process lacks clear specifications, AI assistance amplifies ambiguity. It generates code that solves incorrect problems. It produces technical debt faster than teams can address it.

This whitepaper examines specification quality as the critical variable. It determines AI-assisted development outcomes. We present event modeling and event sourcing as frameworks that provide the specification clarity AI systems need for consistent, correct results.

1.3 Scope and Methodology

This analysis draws on:

- Experience with enterprise implementations using Axon Framework
- Comparative analysis of AI-assisted development with and without formal specifications
- Code quality observations from systems built with event modeling versus traditional approaches

- Developer productivity measurements across multiple organizations

All code examples in this whitepaper are illustrative and simplified for educational purposes. While based on real-world patterns, they should be adapted and thoroughly tested before use in production systems.

2. The Acceleration Problem

2.1 AI as Process Multiplier

AI coding assistants work differently than traditional productivity tools. Static analysis tools identify problems. Build systems automate repetitive tasks. But AI assistants actively generate implementation based on their understanding of requirements.

This creates a multiplicative effect on development processes:

Well-specified processes:

- Clear requirements → AI generates appropriate implementation
- Consistent patterns → AI learns from codebase examples
- Defined boundaries → AI respects architectural constraints

Poorly-specified processes:

- Vague requirements → AI makes incorrect assumptions
- Inconsistent patterns → AI falls back on general knowledge that may not fit
- Undefined boundaries → AI generates code that violates architectural principles

2.2 The “Big Ball of Mud” Acceleration

Consider legacy systems with tangled dependencies, unclear boundaries, and inconsistent architectural decisions. When development teams apply AI coding assistants to these systems:

1. Initial velocity appears high (thousands of lines of code generated daily)
2. Code review becomes a bottleneck (unclear if generated code respects system constraints)
3. Integration reveals problems (generated code violates implicit architectural assumptions)
4. Refactoring costs escalate (volume of generated code requires extensive rework)
5. System complexity compounds (each iteration adds more tangled dependencies)

The AI assistant isn’t malfunctioning. It generates code matching the patterns it observes in the existing system. The problem? Those patterns shouldn’t be replicated.

2.3 Unclear Specifications as Root Cause

Without clear specifications, AI-assisted development encounters systematic problems:

Ambiguity amplification: Vague requirements like "implement user management" leave critical decisions to the AI system. Should passwords be plaintext or hashed? Should validation happen client-side or server-side? Should there be audit logging? The AI must guess.

Context misunderstanding: AI models trained on general code patterns may apply solutions from the wrong context. An e-commerce system's approach to user management differs substantially from a healthcare system's requirements. Without explicit specification, the AI cannot distinguish these contexts.

Inconsistent architectural decisions: Different prompts to the same AI system may produce code using different architectural patterns. This creates a codebase with multiple competing approaches to similar problems.

2.4 The Feedback Loop Problem

Traditional development processes include feedback loops that catch specification problems:

1. Developer interprets requirement
2. Implementation proceeds
3. Code review identifies issues
4. Testing reveals problems
5. User validation confirms or refutes interpretation

These loops have historically compensated for specification ambiguity. The process was slow, but specification errors were eventually discovered and corrected.

AI-assisted development accelerates step 2 dramatically while leaving other steps unchanged. By the time feedback loops identify specification problems, thousands of lines of generated code require review and potential rework. The slow feedback that previously caught specification errors now arrives too late to prevent substantial wasted effort.

3. The Specification Crisis

3.1 The Agile Interpretation Problem

Agile methodology emphasizes “working software over comprehensive documentation.” This created industry-wide skepticism toward upfront specification.

The interpretation addressed legitimate problems with heavyweight documentation processes. But it produced its own pathologies.

Development teams learned to treat “big design upfront” as categorically problematic. The result: systems built without adequate understanding of requirements, relying on iterative discovery through implementation. This approach concealed substantial inefficiencies that slow development velocity made tolerable.

3.2 Why Inadequate Specifications Weren’t Obviously Problematic

Pre-AI development processes absorbed specification ambiguity through:

Developer interpretation: Experienced developers filled specification gaps with reasonable assumptions based on domain knowledge and previous experience.

Incremental discovery: Iterative development allowed requirements clarification through user feedback on working software.

Codebase as documentation: The implementation itself became the primary artifact capturing system behavior, with developers reading existing code to understand how similar problems were solved.

These mechanisms, while inefficient, functioned adequately when human developers implemented all code. Human judgment compensated for specification gaps at every step.

3.3 AI Exposes the Hidden Cost

AI-assisted development removes the human judgment buffer. AI systems:

- Cannot effectively fill specification gaps with reasonable assumptions

- Lack domain knowledge to distinguish appropriate from inappropriate solutions
- Generate code at speeds that prevent iterative discovery from functioning as a specification mechanism

The cost previously hidden in developer time interpreting ambiguous requirements now manifests as generated code requiring extensive review and refactoring.

3.4 The False Innovation of “Specification-Driven Development”

The AI development community has recently embraced specification-driven approaches as though discovering a novel technique. This represents a return to sound engineering fundamentals rather than genuine innovation.

The software industry historically understood specification importance. The abandonment of specification practices reflected their previous high cost and slow feedback loops, not fundamental unsuitability. AI has changed the cost-benefit analysis: poor specifications now create exponentially compounding problems as AI accelerates and amplifies implementation errors at scale.

4. Event Modeling as Specification Language

4.1 The Specification Sweet Spot

Effective specifications for AI-assisted development require precise balance:

Too high-level: “Build user management system” leaves critical decisions undefined, forcing AI to make assumptions.

Too low-level: Pseudocode or detailed implementation steps constrain AI unnecessarily, preventing it from applying its knowledge of language idioms and best practices.

Appropriate level: Clear description of what the system must accomplish, leaving implementation details to the AI while constraining behavior to match requirements.

Event modeling operates at this appropriate level.

4.2 Core Concepts

Event modeling specifies system behavior through slices: atomic units capturing complete workflow steps. Each slice documents:

- **Commands:** The input that triggers behavior
- **Events:** What happens as a result
- **State changes:** How the system’s data evolves
- **Views:** What information becomes available to users or other systems

This isn’t merely documentation. It’s a precise specification of system behavior that bridges business intent and technical implementation.

4.3 Specification Example: User Registration

Consider user registration in an event-modeled system:

```
Slice: User Registration

Command: RegisterUserCommand
  - email: String (validated format, uniqueness check required)
  - password: String (minimum 12 characters, complexity requirements)
  - consentTimestamp: Instant (required, must be recent)

Events Produced:
  - UserRegisteredEvent
    - userId: UUID (generated)
    - email: String (validated)
    - registrationTimestamp: Instant
  - EmailVerificationRequestedEvent
    - userId: UUID
    - verificationToken: String (generated)
    - expiresAt: Instant

State Changes:
  - User entity created (pending verification status)
  - Verification token stored with expiration

Views:
  - ConfirmationEmailView (contains verification link)
  - RegistrationSuccessView (shown to user)
```

This specification is:

Unambiguous: Each element has clear type and validation requirements.

Complete: All inputs, outputs, and state changes are defined.

Implementation-agnostic: The specification doesn't dictate how password hashing or token generation occurs, allowing AI to apply appropriate solutions.

Business-aligned: Domain experts can validate that this correctly captures registration requirements.

4.4 AI-Generated Implementation

With this specification, AI can generate appropriate implementation. Using Axon Framework 5:

```java
// Command definition
@Command(routingKey = "userId")
public record RegisterUserCommand(
    String userId,
    @NotNull @Email String email,
    @NotNull @Size(min = 12) String password,
    @NotNull @PastOrPresent Instant consentTimestamp
) {}

// Event-sourced entity
@EventSourcedEntity
public class User {

    private String userId;
    private String email;
    private UserStatus status;

    @EventSourcingHandler
    private void on(UserRegisteredEvent event) {
        this.userId = event.userId();
        this.email = event.email();
        this.status = UserStatus.PENDING_VERIFICATION;
    }

    @EntityCreator
    protected User() {
        // Required no-arg constructor for reconstitution
    }

    public String getUserId() {
        return userId;
    }

    public UserStatus getStatus() {
        return status;
    }
}

// Stateless command handler
@Component
public class UserCommandHandlers {
```

```
42
43      private final PasswordEncoder passwordEncoder;
44      private final EmailValidator emailValidator;
45
46      public UserCommandHandlers(PasswordEncoder passwordEncoder,
47                                 EmailValidator emailValidator) {
48          this.passwordEncoder = passwordEncoder;
49          this.emailValidator = emailValidator;
50      }
51
52      @CommandHandler
53      public String handle(RegisterUserCommand command,
54                           EventAppender eventAppender) {
55
56          // Validate email format and uniqueness
57          if (!emailValidator.isValid(command.email())) {
58              throw new IllegalArgumentException("Invalid email format");
59          }
60
61          if (emailValidator.exists(command.email())) {
62              throw new IllegalArgumentException("Email already
63  registered");
64          }
65
66          // Validate password complexity
67          if (!meetsComplexityRequirements(command.password())) {
68              throw new IllegalArgumentException(
69                  "Password does not meet complexity requirements"
70              );
71          }
72
73          // Validate consent timestamp is recent (within last hour)
74          if (command.consentTimestamp().isBefore(
75                  Instant.now().minus(Duration.ofHours(1)))) {
76              throw new IllegalArgumentException("Consent timestamp too
77  old");
78          }
79
80          // Generate verification token
            String verificationToken = UUID.randomUUID().toString();
            Instant expiresAt = Instant.now().plus(Duration.ofDays(1));
```

```

        // Append events
        eventAppender.append(new UserRegisteredEvent(
            command.userId(),
            command.email(),
            passwordEncoder.encode(command.password()),
            Instant.now()
        ));

        eventAppender.append(new EmailVerificationRequestedEvent(
            command.userId(),
            verificationToken,
            expiresAt
        ));

        return command.userId();
    }

    private boolean meetsComplexityRequirements(String password) {
        return password.length() >= 12 &&
               password.chars().anyMatch(Character::isUpperCase) &&
               password.chars().anyMatch(Character::isDigit) &&
               password.chars().anyMatch(ch -> !
Character.isLetterOrDigit(ch));
    }
}

// Event definitions
public record UserRegisteredEvent(
    String userId,
    String email,
    String hashedPassword,
    Instant registrationTimestamp
) {}

public record EmailVerificationRequestedEvent(
    String userId,
    String verificationToken,
    Instant expiresAt
) {}
```

```java
// Projection for query side
@Component
@ProcessingGroup("user-projections")
public class UserProjection {

    private final UserViewRepository repository;

    @EventHandler
    public void on(UserRegisteredEvent event) {
        repository.save(new UserView(
            event.userId(),
            event.email(),
            UserStatus.PENDING_VERIFICATION,
            event.registrationTimestamp()
        ));
    }

    @QueryHandler
    public UserView handle(FindUserByEmailQuery query) {
        return repository.findByEmail(query.email())
            .orElseThrow(() -> new UserNotFoundException(query.email()));
    }
}
```

The AI-generated code:

- Implements specified validation (email format, uniqueness, password complexity)
- Produces specified events in correct sequence
- Creates appropriate state changes
- Handles both command and query concerns
- Applies Axon Framework patterns correctly

The specification was unambiguous enough that AI could generate correct implementation.

5. Pattern Repetition and AI Training

5.1 The Cognitive Load of Abundant Choice

Software systems often employ many different structural approaches at the architectural level. One service uses CQRS. Another uses traditional CRUD. One module organizes around aggregates. Another around transactions. One component follows event sourcing. Another uses active record.

This variety creates a hidden cost: cognitive overhead. It affects both human developers and AI systems.

Every architectural decision requires understanding context, evaluating tradeoffs, and selecting appropriate approaches. In systems employing many different architectural patterns sporadically, developers must maintain mental models of all approaches. They must understand when each applies. AI systems face similar challenges. Insufficient examples of each pattern prevent statistical learning from observed code.

This is distinct from implementation patterns like the Gang of Four design patterns (Strategy, Factory, Observer, etc.). Those patterns remain valuable within implementations. The challenge lies at the architectural level—the fundamental structure of how the system processes requests, manages state, and coordinates operations.

5.2 The Event Modeling Approach: Constrained Repetition

Event modeling inverts this paradigm. Instead of many different architectural approaches applied occasionally across different parts of the system, event-driven systems employ four patterns repeatedly:

1. **Command handling pattern:** Validate input, make decision, produce events
2. **Event projection pattern:** React to events, update read models
3. **Query handling pattern:** Retrieve information from read models
4. **Process automation pattern:** React to events, issue commands

These patterns appear hundreds or thousands of times across a system. Each business capability follows the same fundamental structure, varying only in business logic details.

5.3 Pattern Repetition Provides Clear Guidelines

When AI encounters a codebase, its effectiveness depends on recognizable patterns:

Sparse architectural patterns (many different structural approaches, few examples each):

- AI has insufficient examples to understand system-specific approaches
- Falls back on general training from public repositories
- Applies conventional wisdom that may not fit your architecture
- Generates plausible-looking code with subtle incompatibilities

Dense architectural patterns (consistent structural approach, hundreds of examples):

- AI has clear, consistent examples showing your system's structure
- Pattern repetition provides unambiguous guidelines on what to generate
- System-specific approaches override general knowledge
- Generated code matches established patterns

5.4 Practical Example: Command Handler Pattern

Consider an Axon Framework codebase with 200 different command handlers. Each follows identical command handling structure:

```java
// Event-sourced entity with state
@EventSourcedEntity
public class Order {

    private String orderId;
    private OrderStatus status;
    private List<OrderLine> items;

    @EventSourcingHandler
    private void on(OrderCreatedEvent event) {
        this.orderId = event.orderId();
        this.status = OrderStatus.DRAFT;
        this.items = new ArrayList<>(event.items());
    }

    @EventSourcingHandler
    private void on(ItemAddedToOrderEvent event) {
        this.items.add(new OrderLine(
            event.productId(),
            event.quantity(),
            event.unitPrice()
        ));
    }

    @EntityCreator
    protected Order() {
        // Required no-arg constructor
    }

    public String getOrderId() {
        return orderId;
    }

    public OrderStatus getStatus() {
        return status;
    }

    public List<OrderLine> getItems() {
        return items;
    }
}
```

```java

// Stateless command handlers
@Component
public class OrderCommandHandlers {

    @CommandHandler
    public String handle(CreateOrderCommand command,
                         EventAppender eventAppender) {
        // Validation
        if (command.items().isEmpty()) {
            throw new IllegalArgumentException("Order must contain
items");
        }

        // Business logic
        BigDecimal total = calculateTotal(command.items());

        // Event production
        eventAppender.append(new OrderCreatedEvent(
            command.orderId(),
            command.customerId(),
            command.items(),
            total,
            Instant.now()
        ));

        return command.orderId();
    }

    @CommandHandler
    public void handle(AddItemToOrderCommand command,
                       @InjectEntity Order order,
                       EventAppender eventAppender) {
        // Guard clause
        if (order.getStatus() != OrderStatus.DRAFT) {
            throw new IllegalStateException("Cannot modify non-draft
order");
        }

        // Business logic
        if (order.getItems().stream().anyMatch(item ->
```

```
                item.productId().equals(command.productId()))) {
            throw new IllegalArgumentException("Item already in order");
        }

        // Event production
        eventAppender.append(new ItemAddedToOrderEvent(
            order.getOrderId(),
            command.productId(),
            command.quantity(),
            command.unitPrice()
        ));
    }

    private BigDecimal calculateTotal(List<OrderItem> items) {
        return items.stream()
            .map(item →
item.unitPrice().multiply(BigDecimal.valueOf(item.quantity())))
            .reduce(BigDecimal.ZERO, BigDecimal::add);
    }
}
```

This pattern repeats across every command handler: ShippingCommandHandlers, PaymentCommandHandlers, InventoryCommandHandlers, CustomerCommandHandlers, etc. When you ask AI to generate a new handler:

“Create command handlers for subscription management”

The AI can reference 200 similar examples. It understands:

- Creational command handlers don’t receive entity parameters
- Instance command handlers receive entities via @InjectEntity
- Guard clauses validate business rules
- Event sourcing handlers update entity state
- Events are published via EventAppender
- Events are the source of truth

The AI doesn’t need to understand abstract base classes or figure out which interfaces to implement. It follows the established pattern.

6. Purpose-Built Models vs. Shared Abstractions

6.1 The Traditional Abstraction Approach

Object-oriented programming emphasizes reusability through abstraction. Conventional wisdom suggests: if three parts of your system work with customer data, create a single, reusable Customer model. This approach appears to eliminate duplication and promote consistency.

In practice, this creates problematic coupling:

```
01  // Traditional "unified" Customer model (problematic approach)
02  @Entity
03  @Table(name = "customers")
04  public class Customer {
05
06      @Id
07      private String customerId;
08
09      // Billing context needs these
10      @OneToMany(mappedBy = "customer", fetch = FetchType.EAGER)
11      private List<Payment> paymentHistory;
12      private Integer creditScore;
13
14      // Shipping context needs these
15      @OneToMany(mappedBy = "customer", fetch = FetchType.EAGER)
16      private List<Address> addresses;
17      private DeliveryPreferences deliveryPreferences;
18
19      // Marketing context needs these
20      @OneToMany(mappedBy = "customer", fetch = FetchType.EAGER)
21      private List<Purchase> purchaseHistory;
22      private CommunicationPreferences communicationPreferences;
```

```
    // Analytics context needs these
    @OneToMany(mappedBy = "customer", fetch = FetchType.EAGER)
    private List<CustomerEvent> activityLog;
    private SegmentationData segmentation;

    // Getters, setters, business logic that serves all contexts...
}
```

This creates several problems:

Unnecessary coupling: Changes to billing requirements ripple into shipping code. Marketing team modifications affect payment processing. Every context carries the weight of every other context's requirements.

Performance degradation: Each query loads unnecessary data. Billing operations fetch shipping addresses. Marketing queries load payment history. Database joins multiply across unrelated tables.

Cognitive overhead: Developers must understand which fields matter in which contexts. New team members struggle to determine what "Customer" means for their specific use case.

Testing complexity: Unit testing requires mocking or stubbing irrelevant portions of the model. Integration tests must account for cross-context dependencies.

6.2 Event-Driven Approach: Purpose-Built Models

Event modeling inverts this. Each bounded context builds purpose-specific projections:

```java
// Billing context
@Component
@ProcessingGroup("billing-projections")
public class BillingCustomerProjection {

    @EventHandler
    public void on(CustomerRegisteredEvent event) {
        repository.save(new BillingCustomer(
            event.customerId(),
            event.email(),
            new ArrayList<>(), // payment history
            calculateInitialCreditScore(event)
        ));
    }

    @EventHandler
    public void on(PaymentProcessedEvent event) {
        BillingCustomer customer =
repository.findById(event.customerId())
            .orElseThrow();

        customer.addPayment(new Payment(
            event.paymentId(),
            event.amount(),
            event.processedAt()
        ));

        customer.updateCreditScore(calculateCreditScore(customer));
        repository.save(customer);
    }

    @QueryHandler
    public BillingCustomer handle(GetBillingCustomerQuery query) {
        return repository.findById(query.customerId())
            .orElseThrow(() -> new
CustomerNotFoundException(query.customerId()));
    }
}

// Shipping context
@Component
```

```
@ProcessingGroup("shipping-projections")
public class ShippingCustomerProjection {

    @EventHandler
    public void on(CustomerRegisteredEvent event) {
        repository.save(new ShippingCustomer(
            event.customerId(),
            event.email(),
            new ArrayList<>(), // addresses
            DeliveryPreferences.standard()
        ));
    }

    @EventHandler
    public void on(AddressAddedEvent event) {
        ShippingCustomer customer =
repository.findById(event.customerId())
            .orElseThrow();

        customer.addAddress(new ShippingAddress(
            event.addressId(),
            event.street(),
            event.city(),
            event.postalCode(),
            event.country()
        ));

        repository.save(customer);
    }

    @QueryHandler
    public ShippingCustomer handle(GetShippingCustomerQuery query) {
        return repository.findById(query.customerId())
            .orElseThrow(() -> new
CustomerNotFoundException(query.customerId()));
    }
}

// Marketing context
@Component
@ProcessingGroup("marketing-projections")
```

```
public class MarketingCustomerProjection {

    @EventHandler
    public void on(CustomerRegisteredEvent event) {
        repository.save(new MarketingCustomer(
            event.customerId(),
            event.email(),
            new ArrayList<>(), // purchase history
            CommunicationPreferences.defaultPreferences(),
            calculateInitialSegment(event)
        ));
    }

    @EventHandler
    public void on(OrderCompletedEvent event) {
        MarketingCustomer customer =
repository.findById(event.customerId())
            .orElseThrow();

        customer.addPurchase(new Purchase(
            event.orderId(),
            event.totalAmount(),
            event.items(),
            event.completedAt()
        ));

        customer.updateSegmentation(calculateSegment(customer));
        repository.save(customer);
    }

    @QueryHandler
    public MarketingCustomer handle(GetMarketingCustomerQuery query) {
        return repository.findById(query.customerId())
            .orElseThrow(() -> new
CustomerNotFoundException(query.customerId()));
    }
}
```

6.3 Benefits of Purpose-Built Models

This approach delivers substantial advantages:

Clarity: Each model is simple, focused, immediately understandable within its context. Developers working in billing don't encounter shipping concepts. Marketing team members don't navigate payment processing complexity.

Independence: Changes to billing requirements don't affect shipping code. Marketing features evolve independently. Each bounded context proceeds at its own pace without cross-context coordination overhead.

Performance: Each projection contains exactly the data needed for its queries, pre-shaped for efficient access. No unnecessary joins. No loading unused data. Query performance is predictable and optimized.

Testability: You can validate a single projection's behavior without understanding the entire system. Unit tests focus on specific event handling. Integration tests verify individual bounded contexts independently.

AI-friendliness: When generating code for a new projection, AI references dozens of similar projections. Each demonstrates the same pattern in different business contexts. The AI understands: event arrives, state updates, query returns shaped data.

6.4 Addressing the Duplication Concern

Critics observe that purpose-built models duplicate information. A CustomerId appears in billing, shipping, and marketing contexts. Email addresses are stored multiple times. Isn't this wasteful?

This perspective misunderstands the nature of the duplication. The models aren't duplicating the same information. They're maintaining different information that happens to share some identifiers:

- Billing cares about payment history and creditworthiness
- Shipping cares about delivery addresses and preferences
- Marketing cares about purchase patterns and communication preferences

These are fundamentally different concerns. The shared customer ID is merely a correlation key, not evidence that the models should be unified.

The duplication provides value:

Autonomy: Each context evolves independently. Schema changes in one context don't require coordination with others.

Resilience: Failures in one context don't cascade to others. Billing database issues don't prevent shipping operations.

Scalability: Each context can be scaled independently based on its specific load characteristics.

7. Event Sourcing and Complete Context

7.1 State-Centric Systems: The Information Loss Problem

Most modern systems exhibit state-obsession. Databases store current state. When state changes, previous state is overwritten:

```sql
-- Traditional state-centric approach
CREATE TABLE users (
    user_id UUID PRIMARY KEY,
    email VARCHAR(255) NOT NULL,
    address VARCHAR(500),
    email_verified BOOLEAN,
    status VARCHAR(50),
    updated_at TIMESTAMP
);

-- When user moves to new address
UPDATE users
SET address = '456 New Street',
    updated_at = NOW()
WHERE user_id = 'user-123';

-- Previous address information is lost forever
```

This approach appears straightforward. Current state is what matters, right? Consider what information disappears:

When did the user live at the previous address? Unknown. Why did the address change? Unrecorded. How many times has the address changed? Cannot determine. What was the sequence of addresses? Lost.

Real business processes require this historical context:

- Mail forwarding needs the previous address
- School districts need to know when the move occurred

- Insurance companies need both addresses during transition
- Support teams need to understand the timeline of changes
- Analytics teams need to identify patterns in customer behavior

State-centric systems throw away precisely the information business processes need.

7.2 Event Sourcing: Capturing Complete History

Event sourcing inverts the paradigm. Instead of storing current state, systems store the sequence of events that produced that state:

```
// Event sourcing approach - event store schema
CREATE TABLE domain_events (
    event_id UUID PRIMARY KEY,
    aggregate_id UUID NOT NULL,
    aggregate_type VARCHAR(100) NOT NULL,
    event_type VARCHAR(100) NOT NULL,
    event_payload JSONB NOT NULL,
    event_timestamp TIMESTAMPTZ NOT NULL,
    sequence_number BIGINT NOT NULL,
    metadata JSONB,
    UNIQUE(aggregate_id, sequence_number)
);

CREATE INDEX idx_aggregate_events
ON domain_events(aggregate_id, sequence_number);

CREATE INDEX idx_event_type
ON domain_events(event_type);

CREATE INDEX idx_event_timestamp
ON domain_events(event_timestamp);
```

Events capture what actually happened:

```
// Event definitions
public record UserRegisteredEvent(
    String userId,
    String email,
    Instant registeredAt
) {}

public record EmailVerifiedEvent(
    String userId,
    String email,
    Instant verifiedAt
) {}

public record AddressChangedEvent(
    String userId,
    String previousAddress,
    String newAddress,
    Instant changedAt,
    String reason
) {}

public record UserStatusChangedEvent(
    String userId,
    UserStatus previousStatus,
    UserStatus newStatus,
    Instant changedAt,
    String reason
) {}
```

With event sourcing, the event log for user-123 might contain:

```
[
    UserRegisteredEvent(
        userId: "user-123",
        email: "user@example.com",
        registeredAt: 2024-01-01T10:00:00Z
    ),
    EmailVerifiedEvent(
        userId: "user-123",
        email: "user@example.com",
        verifiedAt: 2024-01-01T10:15:00Z
    ),
    AddressChangedEvent(
        userId: "user-123",
        previousAddress: "123 Old Street, City A",
        newAddress: "456 New Street, City B",
        changedAt: 2024-06-15T14:30:00Z,
        reason: "Relocation for employment"
    ),
    UserStatusChangedEvent(
        userId: "user-123",
        previousStatus: ACTIVE,
        newStatus: PREMIUM,
        changedAt: 2024-07-01T09:00:00Z,
        reason: "Upgraded subscription"
    )
]
```

7.3 Projecting Events into Views

Current state isn't abandoned. It's derived from events:

```java
// Current state projection
@Component
@ProcessingGroup("user-current-state")
public class CurrentUserStateProjection {

    private final UserRepository repository;

    @EventHandler
    public void on(UserRegisteredEvent event) {
        repository.save(new UserView(
            event.userId(),
            event.email(),
            null, // no address yet
            false, // email not verified
            UserStatus.PENDING_VERIFICATION,
            event.registeredAt()
        ));
    }

    @EventHandler
    public void on(EmailVerifiedEvent event) {
        UserView user = repository.findById(event.userId())
            .orElseThrow();

        user.setEmailVerified(true);
        user.setStatus(UserStatus.ACTIVE);
        repository.save(user);
    }

    @EventHandler
    public void on(AddressChangedEvent event) {
        UserView user = repository.findById(event.userId())
            .orElseThrow();

        user.setAddress(event.newAddress());
        repository.save(user);
    }

    @QueryHandler
    public UserView handle(GetCurrentUserStateQuery query) {
        return repository.findById(query.userId())
```

```
            .orElseThrow(() -> new
UserNotFoundException(query.userId()));
    }
}

// Historical address projection (for mail forwarding)
@Component
@ProcessingGroup("user-address-history")
public class AddressHistoryProjection {

    private final AddressHistoryRepository repository;

    @EventHandler
    public void on(AddressChangedEvent event) {
        repository.save(new AddressHistoryEntry(
            event.userId(),
            event.previousAddress(),
            event.newAddress(),
            event.changedAt(),
            event.reason()
        ));
    }

    @QueryHandler
    public List<AddressHistoryEntry> handle(GetAddressHistoryQuery query)
{
        return repository.findByUserId(query.userId())
            .stream()
            .sorted(Comparator.comparing(AddressHistoryEntry::changedAt))
            .collect(Collectors.toList());
    }
}

// Verification timeline projection (for support)
@Component
@ProcessingGroup("user-verification-timeline")
public class VerificationTimelineProjection {

    private final Map<String, VerificationTimeline> timelines =
        new ConcurrentHashMap<>();
```

```
    @EventHandler
    public void on(UserRegisteredEvent event) {
        timelines.put(event.userId(), new VerificationTimeline(
            event.userId(),
            event.registeredAt(),
            null,
            null
        ));
    }

    @EventHandler
    public void on(EmailVerifiedEvent event) {
        VerificationTimeline timeline = timelines.get(event.userId());
        if (timeline != null) {
            timeline.setVerifiedAt(event.verifiedAt());
            timeline.setDaysToVerify(
                Duration.between(
                    timeline.getRegisteredAt(),
                    event.verifiedAt()
                ).toDays()
            );
        }
    }

    @QueryHandler
    public VerificationTimeline handle(GetVerificationTimelineQuery
query) {
        return Optional.ofNullable(timelines.get(query.userId()))
            .orElseThrow(() -> new
UserNotFoundException(query.userId()));
    }
}
```

Each projection serves specific query requirements. Current state projection provides what most UI operations need. Address history projection supports mail forwarding and analytics. Verification timeline projection assists support operations.

All projections derive from the same source of truth: the event log.

7.4 AI Benefits from Complete Context

Large language models function by understanding context and patterns. Event sourcing provides both:

Context richness: State-centric systems give AI a snapshot. Event stores provide a story. The AI can observe not just current state but how the system arrived at that state.

Temporal reasoning: Events include timestamps and sequencing. AI can reason about: timing patterns (how long between registration and verification), behavioral sequences (what typically follows what), anomaly detection (unusual patterns in event sequences).

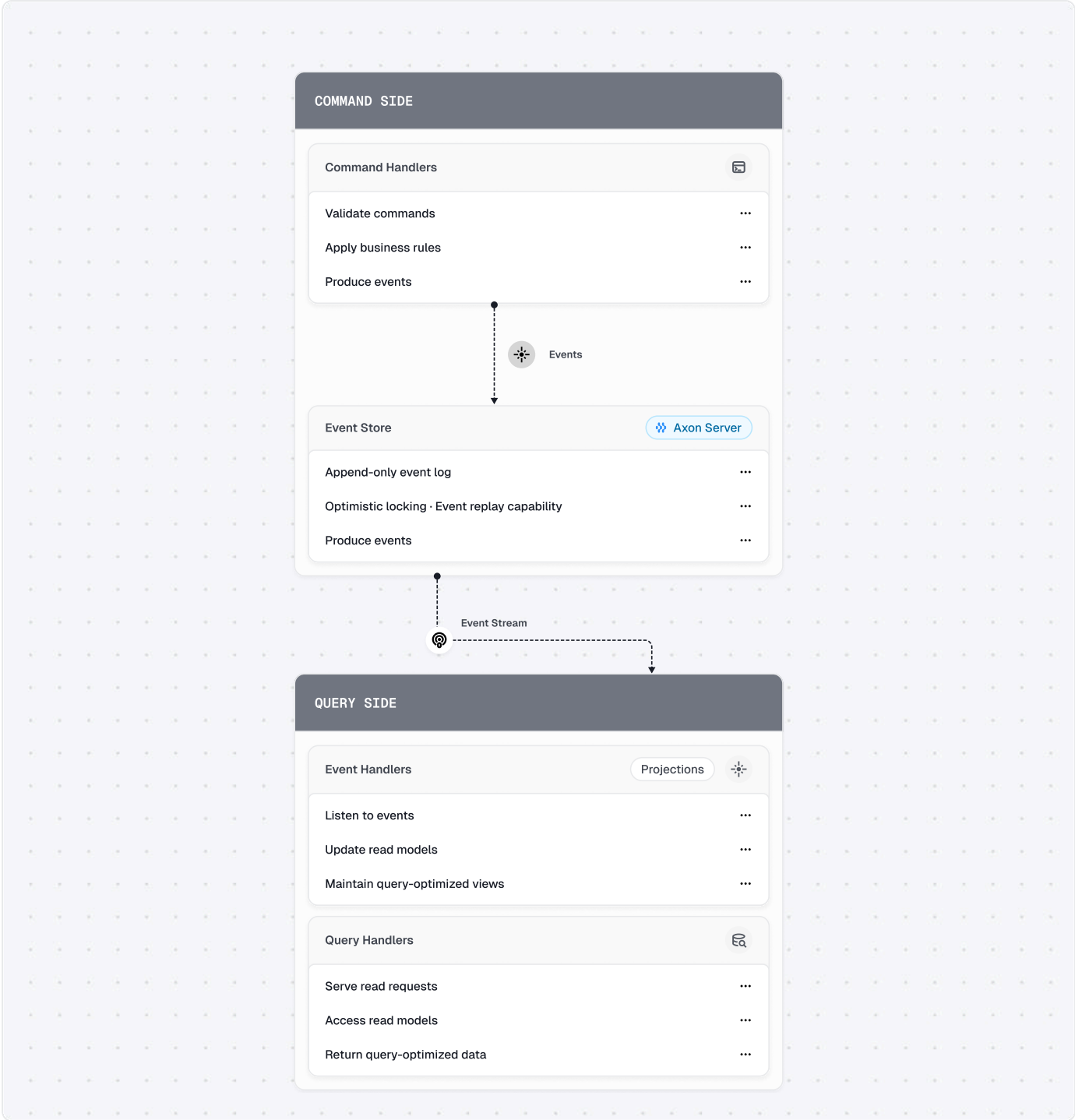
Causality understanding: Events explicitly capture cause and effect. The AI observes that AddressChangedEvent followed by EmailVerificationRequestedEvent might indicate the user needs to reverify from the new location.

When AI generates code in event-sourced systems, it has access to complete behavioral context rather than merely current state. This enables more sophisticated reasoning about appropriate system behavior.

8. Production Implementation

8.1 Architecture Overview

Production event-sourced systems using Axon Framework follow consistent architectural patterns:



8.2 Specification-Driven Development Workflow

The production workflow integrates event modeling with AI-assisted implementation:

Phase 1: Event Modeling Workshop

- Business stakeholders and technical team collaborate
- Identify slices (complete workflow steps)
- Define commands, events, state changes, and views
- Validate business logic with domain experts
- Result: Unambiguous specification

Phase 2: Technical Specification

- Convert event model into structured specifications
- Define validation rules, constraints, and invariants
- Specify data types and formats
- Document integration points
- Result: AI-ready specification

Phase 3: AI-Assisted Implementation

- Use AI to generate boilerplate code from specifications
- Review generated code for specification compliance
- Implement business logic in generated structure
- Result: Working implementation matching specification

Phase 4: Validation and Refinement

- Test generated code against specification
- Validate behavior with domain experts
- Refine based on feedback
- Result: Validated implementation ready for further testing

8.3 Example: Complete Slice Implementation

Consider a complete order processing slice from specification to implementation:

Event Model Specification:

```
Slice: Create Order

Command: CreateOrderCommand
  - orderId: UUID (generated)
  - customerId: UUID (must exist)
  - items: List<OrderItem> (minimum 1 item)
    - productId: UUID (must be valid product)
    - quantity: Integer (positive, within stock limits)
    - unitPrice: BigDecimal (matches current product price)
  - shippingAddress: Address (validated format)

Business Rules:
  - Customer must have verified email
  - All products must be currently available
  - Total order value must not exceed customer credit limit
  - Shipping address must be within serviceable regions

Events Produced:
  - OrderCreatedEvent
    - orderId: UUID
    - customerId: UUID
    - items: List<OrderItem>
    - totalAmount: BigDecimal
    - createdAt: Instant
  - InventoryReservedEvent
    - orderId: UUID
    - reservations: List<InventoryReservation>

State Changes:
  - Order entity created (status: PENDING_PAYMENT)
  - Inventory reserved for order items

Views:
  - OrderConfirmationView (for customer)
  - InventoryReservationView (for warehouse)
```

AI-Generated Implementation:

```java
// Command
@Command(routingKey = "orderId")
public record CreateOrderCommand(
    String orderId,
    @NotNull UUID customerId,
    @NotEmpty List<OrderItem> items,
    @NotNull @Valid Address shippingAddress
) {
    public record OrderItem(
        @NotNull UUID productId,
        @Positive int quantity,
        @NotNull @DecimalMin("0.01") BigDecimal unitPrice
    ) {}
}

// Event-sourced entity
@EventSourcedEntity
public class Order {

    private String orderId;
    private UUID customerId;
    private List<OrderItem> items;
    private BigDecimal totalAmount;
    private OrderStatus status;

    @EventSourcingHandler
    private void on(OrderCreatedEvent event) {
        this.orderId = event.orderId();
        this.customerId = event.customerId();
        this.items = event.items();
        this.totalAmount = event.totalAmount();
        this.status = OrderStatus.PENDING_PAYMENT;
    }

    @EventSourcingHandler
    private void on(InventoryReservedEvent event) {
        // Event recorded, no state change needed in entity
    }

    @EntityCreator
    protected Order() {
```

```
        // Required no-arg constructor for reconstitution
    }

    public String getOrderId() {
        return orderId;
    }

    public OrderStatus getStatus() {
        return status;
    }
}

// Stateless command handler
@Component
public class OrderCommandHandlers {

    private final CustomerRepository customerRepository;
    private final ProductRepository productRepository;
    private final InventoryService inventoryService;
    private final ShippingService shippingService;

    public OrderCommandHandlers(CustomerRepository customerRepository,
                                ProductRepository productRepository,
                                InventoryService inventoryService,
                                ShippingService shippingService) {
        this.customerRepository = customerRepository;
        this.productRepository = productRepository;
        this.inventoryService = inventoryService;
        this.shippingService = shippingService;
    }

    @CommandHandler
    public String handle(CreateOrderCommand command,
                         EventAppender eventAppender) {

        // Validate customer
        Customer customer =
customerRepository.findById(command.customerId())
            .orElseThrow(() -> new
CustomerNotFoundException(command.customerId()));
```

```
        if (!customer.isEmailVerified()) {
            throw new IllegalStateException(
                "Customer email must be verified to place orders"
            );
        }

        // Validate products and calculate total
        BigDecimal total = BigDecimal.ZERO;
        List<InventoryReservation> reservations = new ArrayList<>();

        for (CreateOrderCommand.OrderItem item : command.items()) {
            Product product =
productRepository.findById(item.productId())
                .orElseThrow(() -> new
ProductNotFoundException(item.productId()));

            if (!product.isAvailable()) {
                throw new IllegalStateException(
                    "Product " + item.productId() + " is not available"
                );
            }

            if (!product.getPrice().equals(item.unitPrice())) {
                throw new IllegalArgumentException(
                    "Price mismatch for product " + item.productId()
                );
            }

            if (!inventoryService.canReserve(item.productId(),
item.quantity())) {
                throw new IllegalStateException(
                    "Insufficient inventory for product " +
item.productId()
                );
            }

            BigDecimal itemTotal = item.unitPrice()
                .multiply(BigDecimal.valueOf(item.quantity()));
            total = total.add(itemTotal);

            reservations.add(new InventoryReservation(
```

```
                item.productId(),
                item.quantity()
            ));
        }

        // Validate credit limit
        if (total.compareTo(customer.getCreditLimit()) > 0) {
            throw new IllegalStateException(
                "Order total exceeds customer credit limit"
            );
        }

        // Validate shipping address
        if (!shippingService.isServiceable(command.shippingAddress())) {
            throw new IllegalArgumentException(
                "Shipping address is not in serviceable region"
            );
        }

        // Append events
        eventAppender.append(new OrderCreatedEvent(
            command.orderId(),
            command.customerId(),
            command.items().stream()
                .map(item -> new OrderCreatedEvent.OrderItem(
                    item.productId(),
                    item.quantity(),
                    item.unitPrice()
                ))
                .collect(Collectors.toList()),
            total,
            Instant.now()
        ));

        eventAppender.append(new InventoryReservedEvent(
            command.orderId(),
            reservations
        ));

        return command.orderId();
    }
```

```java
}

// Projections
@Component
@ProcessingGroup("order-projections")
public class OrderProjection {

    private final OrderViewRepository repository;

    @EventHandler
    public void on(OrderCreatedEvent event) {
        repository.save(new OrderView(
            event.orderId(),
            event.customerId(),
            event.items(),
            event.totalAmount(),
            OrderStatus.PENDING_PAYMENT,
            event.createdAt()
        ));
    }

    @QueryHandler
    public OrderView handle(GetOrderQuery query) {
        return repository.findById(query.orderId())
            .orElseThrow(() -> new
OrderNotFoundException(query.orderId()));
    }

    @QueryHandler
    public List<OrderView> handle(GetCustomerOrdersQuery query) {
        return repository.findByCustomerId(query.customerId());
    }
}

@Component
@ProcessingGroup("inventory-projections")
public class InventoryReservationProjection {

    private final ReservationRepository repository;

    @EventHandler
```

```
    public void on(InventoryReservedEvent event) {
        for (InventoryReservation reservation : event.reservations()) {
            repository.save(new ReservationView(
                event.orderId(),
                reservation.productId(),
                reservation.quantity(),
                ReservationStatus.ACTIVE,
                Instant.now()
            ));
        }
    }

    @QueryHandler
    public List<ReservationView> handle(GetOrderReservationsQuery query)
{
        return repository.findByOrderId(query.orderId());
    }
}
```

8.4 Code Quality Characteristics

The generated code exhibits several desirable properties:

Specification compliance: All validation rules from the specification are implemented. Business rules are enforced. Events match specified structure.

Pattern consistency: Command handling follows established handler pattern. Event handlers follow projection pattern. Query handlers follow query pattern.

Maintainability: Each component has single responsibility. Business logic is clearly expressed. Dependencies are explicit.

Testability: Command handler logic can be tested with command input and event output verification. Projections can be tested with event input and query output verification.

8.5 Configuration and Infrastructure

Real-world systems require appropriate configuration:

```java
// Axon configuration
@Configuration
public class AxonConfig {

    @Bean
    public Configurer configurer(
            EventStore eventStore,
            TokenStore tokenStore) {

        return DefaultConfigurer.defaultConfiguration()
            .configureEmbeddedEventStore(c -> eventStore)
            .configureTokenStore(c -> tokenStore)
            .configureEventSerializer(c -> JacksonSerializer.builder()
                .objectMapper(objectMapper())
                .build())
            .configureMessageSerializer(c -> JacksonSerializer.builder()
                .objectMapper(objectMapper())
                .build());
    }

    @Bean
    public EventStore eventStore(AxonServerConnectionManager
connectionManager,
                                 AxonServerConfiguration configuration) {
        return AxonServerEventStore.builder()
            .platformConnectionManager(connectionManager)
            .configuration(configuration)
            .build();
    }

    @Bean
    public TokenStore tokenStore(DataSource dataSource) {
        return JdbcTokenStore.builder()
            .connectionProvider(new
UnitOfWorkAwareConnectionProviderWrapper(
                new DataSourceConnectionProvider(dataSource)))
            .schema(TokenSchema.defaultInstance())
            .build();
    }

    private ObjectMapper objectMapper() {
```

```
        ObjectMapper mapper = new ObjectMapper();
        mapper.registerModule(new JavaTimeModule());
        mapper.disable(SerializationFeature.WRITE_DATES_AS_TIMESTAMPS);
        return mapper;
    }
}
```

9. Cost Analysis

9.1 Without Clear Specifications

Initial Phase:

- High initial velocity (AI generates code rapidly)
- Minimal specification overhead
- Rapid feature implementation

Development Phase:

- Extended code review (3-4x normal duration due to unclear intent)
- Frequent refactoring (assumptions proved incorrect)
- Technical debt accumulation
- Performance issues emerge
- Security vulnerabilities discovered

Maintenance Phase:

- Decreasing velocity (complexity compounds)
- Increasing bug rate (interactions not fully understood)
- Difficult onboarding (no clear system model)
- Expensive modifications (unclear boundaries and dependencies)

Total Cost Profile:

- Low upfront investment
- Exponentially increasing maintenance costs
- Decreasing business agility over time

9.2 With Event Modeling and Event Sourcing

Initial Phase:

- Moderate initial velocity (specification work required)
- Days to weeks of event modeling workshops

- Structured problem understanding

Development Phase:

- Rapid code review (generated code matches specification)
- Minimal refactoring (design validated upfront)
- Controlled technical debt
- Predictable performance
- Security validated in specification

Maintenance Phase:

- Sustained velocity (clear patterns and boundaries)
- Low bug rate (behavior well-defined)
- Efficient onboarding (clear system model)
- Manageable modifications (explicit dependencies)

Total Cost Profile:

- Higher upfront investment
- Stable maintenance costs
- Maintained business agility over time

9.3 Observed Patterns from Enterprise Customers

Based on observations and feedback from enterprise customers working with event-driven architectures:

Code Review Duration: Customers report that code reviews for AI-generated code take substantially less time when working from clear specifications. Where teams previously spent most of a day reviewing a feature implementation, they now spend a few hours. The difference appears most pronounced in complex business logic where ambiguity would otherwise require extensive back-and-forth.

Bug Rate (early months): We observe significantly fewer bugs in AI-generated code when specifications are clear. Teams working without detailed specifications report finding issues regularly during initial testing phases. Teams working from event models report finding occasional issues, typically at integration points rather than in core business logic.

Onboarding Time: New team members appear to reach productivity much faster when event models document system behavior. Where teams previously reported months for new developers to understand the system well enough to contribute confidently, teams with event models report weeks. The explicit documentation of “what happens when” seems to eliminate much of the learning curve.

Modification Cost (after months of operation): Changes to existing functionality cost substantially more in systems without clear specifications. Teams report that modifications often require extensive code archaeology to understand current behavior before making changes. With event sourcing and clear specifications, the history is explicit and modifications are more straightforward. Customers indicate that changes typically cost a fraction more than the original implementation rather than several times more.

9.4 When Specification Overhead Is Justified

Event modeling with event sourcing makes economic sense when:

Long-term maintenance is expected: Systems that will be maintained for years benefit from clear specifications.

Business logic is complex: Domains with intricate rules and workflows justify upfront specification investment.

Team size is substantial: Larger teams benefit more from shared understanding through explicit specifications.

Compliance requirements exist: Regulated industries require auditable behavior documentation.

AI-assisted development is standard practice: The quality difference between specified and unspecified AI-generated code increases with system complexity.

Event modeling may not be economically justified for:

Short-lived prototypes: Systems expected to be rewritten or discarded don’t benefit from long-term maintainability.

Genuinely simple systems: Systems with minimal business logic don’t require elaborate specification frameworks.

Small teams with strong tacit knowledge: Two developers who fully understand the problem space may not need explicit specifications.

10. Conclusions and Recommendations

10.1 Key Findings

AI coding assistants represent a fundamental shift in software development economics. These tools function as process multipliers. They accelerate whatever development approach they encounter. This makes specification quality the critical variable. It determines whether AI assistance delivers genuine productivity gains or accelerates technical debt accumulation.

Traditional development processes absorbed specification ambiguity through human judgment at every implementation step. AI removes this buffer. It exposes the hidden costs of inadequate specifications. The slow feedback loops that previously caught specification problems now arrive too late to prevent substantial wasted effort.

Event modeling provides a specification framework well-suited for AI-assisted development. It operates at the appropriate level of abstraction. Detailed enough for implementation. High-level enough for stakeholder validation. Event modeling produces unambiguous specifications that enable AI systems to generate correct implementations.

Pattern repetition in event-driven architectures gives AI clear, consistent examples to follow when generating code. Rather than many different architectural approaches applied sporadically, event-driven systems employ four patterns repeatedly. Hundreds or thousands of implementations of the same patterns. With a limited number of patterns repeated consistently, AI has clear guidelines on what to implement. There is little left to hallucinate about. The AI doesn't fall back on general knowledge that may not fit your architecture.

Event sourcing's complete historical context enables more effective AI reasoning about system behavior. Instead of current state snapshots, AI systems have access to complete behavioral stories. This enables temporal reasoning, causality understanding, and pattern recognition. All of which improves generated code quality.

10.2 Practical Recommendations

For Organizations Adopting AI-Assisted Development:

1. **Invest in specification practices:** The cost of specification development is substantially lower than the cost of reviewing and refactoring AI-generated code that solved the wrong problem.
2. **Evaluate event modeling:** For systems where long-term maintainability matters, event modeling provides proven specification approaches compatible with AI-assisted development.
3. **Prioritize pattern consistency:** Establishing repeatable patterns throughout your codebase improves AI-generated code quality more than sophisticated abstractions.
4. **Measure the right metrics:** Optimize for code quality and maintainability rather than raw generation velocity.

For Development Teams:

1. **Resist the temptation to skip specification:** Initial velocity gains from immediate code generation create technical debt that compounds rapidly.
2. **Learn event modeling techniques:** Even if not implementing full event sourcing, event modeling's specification approaches improve AI-assisted development outcomes.
3. **Build purpose-specific models:** Resist the urge to create shared abstractions. Purpose-built models improve clarity, independence, and AI-friendliness.
4. **Leverage AI for appropriate tasks:** Use AI for implementation details and boilerplate generation, not for fundamental architectural decisions.

For Technical Leaders:

1. **Set specification standards:** Establish clear expectations for specification quality before AI-assisted implementation begins.
2. **Create pattern libraries:** Document and enforce consistent patterns that AI systems can learn from.
3. **Monitor technical debt:** Track indicators of specification quality problems before they compound.
4. **Budget for specification work:** Allocate time for event modeling workshops and specification development in project plans.

10.3 Future Directions

AI-assisted development will continue evolving. Several trends warrant attention:

Improved context handling: As AI models' context windows expand, they'll better handle complex system specifications. This strengthens the case for comprehensive event modeling.

Agentic systems: More autonomous AI systems will require even clearer specifications and guardrails. Event modeling's explicit boundaries become increasingly important.

Specification as code: Tools that represent event models in machine-readable formats will enable more sophisticated AI assistance.

Pattern learning: AI systems that actively learn from your codebase patterns will benefit more from consistent event-driven architectures.

10.4 Final Assessment

The productivity paradox of AI-assisted development resolves when we recognize a simple truth: these tools optimize whatever we ask them to optimize.

Without clear specifications, they optimize code generation velocity. At the expense of correctness and maintainability.

With clear specifications, they optimize implementation of well-defined behavior.

Event modeling and event sourcing don't make AI coding assistants necessary. They make them productive. The combination works: human domain expertise (captured in event models) plus AI implementation capability (guided by clear specifications). This delivers genuine productivity improvements without sacrificing code quality or long-term maintainability.

For organizations serious about AI-assisted development, specification practices aren't optional overhead. They're the foundation. They determine whether AI assistance accelerates value delivery or technical debt accumulation.

The choice is clear: invest in specifications upfront, or pay exponentially more in maintenance costs later.

References

1. Fowler, M. (2005). Event Sourcing. MartinFowler.com. Available at: https://martinfowler.com/eaaDev/EventSourcing.html
2. Dymitruk, A. (2020). Event Modeling: What is it? EventModeling.org. Available at: https://eventmodeling.org/posts/what-is-event-modeling/
3. Dilger. (2024). Understanding Event Sourcing: Planning and Implementing Scalable Systems with Event Modeling and Event Sourcing. Available at: (https://leanpub.com/eventmodeling-and-eventsourcing)
4. Young, G. (2010). CQRS and Event Sourcing. Retrieved from various conference presentations and documentation.
5. Vernon, V. (2013). Implementing Domain-Driven Design. Addison-Wesley Professional.
6. Axon Framework Documentation (2024). Axon Framework Reference Guide. AxonIQ. Available at: https://docs.axoniq.io/
7. Evans, E. (2003). Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley Professional.
8. Stopford, B. (2018). Designing Event-Driven Systems. O'Reilly Media.
9. Kleppmann, M. (2017). Designing Data-Intensive Applications. O'Reilly Media.
10. Richardson, C. (2018). Microservices Patterns. Manning Publications.

About the Authors

Allard Buijze is the founder of Axoniq and creator of the Axon Framework. With over 15 years of experience in event-driven architecture, he has guided numerous organizations through event sourcing and CQRS implementations. He regularly speaks at conferences worldwide on topics related to event-driven design, domain-driven design, and software architecture.

Martin Dilger is the founder of Nebulit GmbH and author of Understanding Eventsourcing, the first German-language book on Event Modeling. With over 20 years of experience in software architecture, he has helped teams across the DACH region align business and engineering, close requirements gaps, and deliver event-driven systems with confidence. He is a recognized core contributor to event modeling and regularly speaks and trains on event-driven design, event sourcing, and CQRS.

Adam Dymitruk is the CEO of Adaptech Group and the creator of event modeling. With over three decades of experience in software development and architecture, Adam is an expert in event-driven systems and event sourcing. He has been instrumental in advancing these methodologies, particularly through his work on event modeling, which provides a visual and collaborative approach to system design. Adam is a frequent speaker at international conferences and has contributed significantly to the evolution of responsible, efficient system design and implementation in the software industry.

About Axoniq

Axoniq provides products and services for building event-driven applications using event sourcing and CQRS. The company's flagship products include Axon Framework, a Java framework for building event-driven applications, and Axon Server, a purpose-built event store and message routing solution.

For more information: https://www.axoniq.io

ABOUT Nebulit

Nebulit helps software teams in Europe close requirements gaps, align business and engineering, and deliver event-driven and event-sourced systems with confidence - through workshops, training, and architecture consulting built on industry proven methods and technologies.

Learn more: https://nebulit.de/en/index.html

ABOUT Adaptech Group

Adaptech Group is a collective of top technologists focusing around microservices and event-centric architecture, involving concepts like event modeling and event sourcing. They provide teams of highly skilled developers that can jump-start projects, services, or products. They provide consultation, outsourcing, and even staffing; following a method of enabling a unified vision across the organization to enable the business and technical staff to use the same language and share the same vision. Adaptech Group embraces the understanding of Conway's Law as a means to offer solutions with the best architecture, enabling scalability concerns to be adjusted easily in the future.

Learn more: https://adaptechgroup.com/

This whitepaper represents analysis and opinions based on real-world experience and established software engineering principles. Code examples are illustrative and simplified for educational purposes. Implementation results may vary based on specific organizational context, team capabilities, and system requirements. Always thoroughly test and adapt approaches based on your specific needs and constraints before deploying to production.