Member-only story

# Building Multi-Agent Systems That Actually Work: A 7-Step Production Guide

How to Ship Production-Ready Multi-Agent Systems Without the Technical Debt

19 min read · 1 day ago

Reza Rezvani



Six months ago, we shipped a multi-agent order processing system to production. It handled inventory, payments, and fulfillment across three independent agents. The POC took two weeks and looked perfect in demos.

multi-agent_system.py

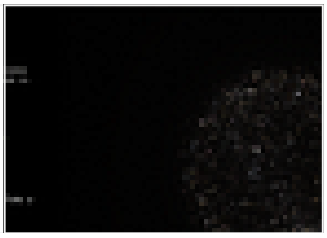
```
import planner_agent
import coder_agent

class MultiAgentSystem:
    def __init__(self):
        self.planner = planner_aget.Plannereλ()
        self.coder = coder_agent.CoderAgent()
        self.deployer = deployer_agent.Deployer

    def run(self, task_description):
        plan = self.planner.plan(task_escŕiption)
        code = self.coder.write_code(plan)
        run-deployer.deploy(

system = MultiAgentSystem()

system.run("Build a web app")
```

Multi Agent System as a PoC

Then we turned on production traffic.

**Within 48 hours:** agents were creating infinite message loops, deadlocking each other waiting for responses, and spawning new instances until we hit resource limits. Our staging environment crashed 17 times in the first week. Mean time to recovery was 45 minutes because we couldn't figure out which agent was causing the failure.





The problem wasn't that multi-agent systems don't work. The problem was that we treated coordination architecture as an afterthought. We focused on making individual agents smart but ignored how they'd work together under production load.

After rebuilding the system with proper coordination infrastructure, we've maintained **99.7% uptime** for six months, handle **10K+ concurrent workflows**, and debug issues in under **15 minutes**. The difference wasn't smarter agents — it was better architecture.

In my 22 years building distributed systems, I've seen this pattern repeat: **90% of multi-agent POCs fail to reach production** not because the agents themselves are complex, but because the coordination layer is fundamentally broken. Teams build agents that work in isolation, then discover they can't make them work together reliably.

This guide covers the seven coordination patterns that separate production-ready multi-agent systems from POCs that crash under load. These aren't theoretical frameworks — they're battle-tested patterns from systems processing millions of transactions.

. . .

## The 7 Critical Failures: What Breaks Multi-Agent Systems in Production

Before we dive into solutions, understand what actually fails. Every multi-agent system that struggles in production hits at least 3–4 of these issues:

1. **No Clear Agent Boundaries**
   Agents with overlapping responsibilities make conflicting decisions. When both the InventoryAgent and OrderAgent try to reserve stock, you get race conditions.

**Symptom:** Duplicate work, inconsistent state, unclear accountability.

**2. Synchronous Communication Deadlocks**

Agent A calls Agent B, which calls Agent C, which needs a response from Agent A.

System freezes.

**Symptom:** Deadlocks, cascading failures, tight coupling between agents.

**3. No State Management Between Interactions**
Agents lose context between messages. They can't remember what happened three steps ago in a multi-step workflow.

**Symptom:** Repeated work, lost progress, orphaned transactions.

**4. Missing Error Recovery Patterns**
One agent crashes and takes down the entire system. No circuit breakers, no fallbacks, no graceful degradation.

**Symptom:** System-wide outages from single agent failures.

**5. No Resource Limits**
Agents spawn new instances infinitely. Message queues grow unbounded. Memory exhaustion crashes production.

**Symptom:** Resource exhaustion, cascading failures, unpredictable performance.

**6. Unclear Ownership and Accountability**
When something fails, you can't determine which agent should have handled it. Multiple agents try to fix the same problem.

**Symptom:** Duplicate compensation logic, unclear SLAs, impossible debugging.

**7. Missing Observability Infrastructure**
You can't trace requests across agents. No correlation IDs. Logs are scattered.

**Symptom:** 45+ minute debugging sessions, impossible root cause analysis, blind deployments.

Each of these failures costs **30–60 days of debugging and rework** if you hit them in production. Fix them upfront with proper architecture, and you ship reliable systems in **14–21 days.**

> The difference isn't agent intelligence. It's coordination infrastructure.

. . .

Multi-Agent Orchestration in AI Assistants

## Step 1: Define Agent Boundaries (Single Responsibility Architecture)

### What & Why This Matters

**The first mistake teams make:** creating agents with fuzzy, overlapping responsibilities. They build a *"UserAgent"* that handles authentication **and** profile management **and** preferences. Then they're surprised when it becomes a bottleneck and debugging nightmare.

**The problem:** When responsibilities overlap, agents make conflicting decisions. Your InventoryAgent thinks stock is available because it just checked. Your OrderAgent thinks it's reserved because it sent a reservation request. Your PaymentAgent starts processing payment while InventoryAgent realizes stock was actually oversold.

**The result:** race conditions, duplicate work, inconsistent state, and complete chaos under concurrent load.

**The solution:** Single Responsibility Per Agent. Each agent has exactly ONE clearly defined job. If you can't describe an agent's responsibility in a single sentence without using "and," you have multiple agents pretending to be one.

**Claude Code 2.0.27:**

Claude Code 2.0.27: Why and How this Update Actually Matters (And Why It Doesn't Replace Your Terminal) Web-based...

alirezarezvani.medium.com


## The Agent Mapping Exercise

Before writing any code, map your agent boundaries. This is architecture work, not implementation work. Get it wrong here, and you'll rebuild the entire system in three months.

```
interface AgentDefinition {
  name: string;
  responsibility: string;  // Single sentence describing ONLY what this agent d
  inputs: string[];        // What data it receives from other agents
  outputs: string[];       // What data it produces for other agents
  dependencies: string[];  // Which agents it depends on (message flow)
  constraints: string[];   // What it explicitly MUST NOT do (prevents overlap)
}

/**
 * Example: E-commerce order processing system
 *
 * This shows proper boundary definition for a real production system.
 * Each agent has a single, clear responsibility with no overlap.
 */
const agentArchitecture: AgentDefinition[] = [
  {
    name: 'InventoryAgent',
    responsibility: 'Checks product availability and creates inventory reservat
    inputs: ['productId', 'quantity', 'customerId'],
    outputs: ['reservationId', 'availabilityStatus', 'reservationExpiry'],
    dependencies: [],
    constraints: [
      'Must NOT handle payment processing',
      'Must NOT update product prices',
      'Must NOT create shipping orders',
      'Must NOT communicate with customers'
    ]
  },
```

```
    {
      name: 'PaymentAgent',
      responsibility: 'Processes payment transactions and handles payment failure
      inputs: ['reservationId', 'paymentMethod', 'amount', 'customerId'],
      outputs: ['transactionId', 'paymentStatus', 'authorizationCode'],
      dependencies: ['InventoryAgent'],
      constraints: [
        'Must NOT modify inventory levels',
        'Must NOT calculate taxes or shipping',
        'Must NOT create fulfillment orders',
        'Must NOT handle inventory expiration'
      ]
    },
    {
      name: 'FulfillmentAgent',
      responsibility: 'Creates and tracks shipping orders for confirmed payments'
      inputs: ['transactionId', 'shippingAddress', 'items', 'priority'],
      outputs: ['shippingId', 'trackingNumber', 'estimatedDelivery'],
      dependencies: ['PaymentAgent'],
      constraints: [
        'Must NOT process refunds',
        'Must NOT contact customers directly',
        'Must NOT modify payment status',
        'Must NOT adjust inventory'
      ]
    },
    {
      name: 'NotificationAgent',
      responsibility: 'Sends customer notifications via email, SMS, and push',
      inputs: ['customerId', 'notificationType', 'templateData'],
      outputs: ['notificationId', 'deliveryStatus', 'deliveryTime'],
      dependencies: ['PaymentAgent', 'FulfillmentAgent'],
      constraints: [
        'Must NOT modify order state',
        'Must NOT retry failed payments',
        'Must NOT make business logic decisions',
        'Must NOT directly query other agents'
      ]
    }
  ];
```

### The Boundary Test

Use this simple test when defining agents: **Can you describe its responsibility in one sentence without using “and”?**

**Common mistakes:**

- ❌ “UserAgent handles authentication **and** profile management”

- ❌ “OrderAgent processes orders **and** sends notifications”
- ❌ “DataAgent fetches data **and** transforms it **and** caches results”

**Correct boundaries:**

- ✅ “AuthAgent validates credentials and issues access tokens”
- ✅ “ProfileAgent manages user profile data”
- ✅ “OrderAgent coordinates the order workflow state machine”
- ✅ “NotificationAgent sends messages via configured channels”

### Expected Results After This Step

When you’ve properly defined agent boundaries:

- **5–15 agents** for most production systems (not 3, not 30)
- **Zero responsibility overlap** between agents
- **Clear data flow** with defined inputs/outputs
- **Obvious ownership** — when something fails, you know which agent should handle it
- **Independent deployment** — can update one agent without touching others
- **Testable in isolation** — can test each agent with mocked dependencies

If you have 30+ agents, you’re probably splitting too fine. If you have 2–3 agents, they’re probably doing too much. The sweet spot for most systems is 8–12 clearly defined agents.

. . .

## Step 2: Design Async Message Passing (Event-Driven Communication)

### What & Why Synchronous Communication Fails

Here’s what happens with synchronous agent communication:

Agent A needs something from Agent B. It calls Agent B and waits. Agent B needs something from Agent C, so it calls Agent C and waits. Agent C needs to query Agent A to complete its work. **Deadlock.**

**Or worse:** *Agent A calls Agent B. Agent B is temporarily down. Agent A fails. Agent A was handling a critical order. Order lost.*

**The problems with synchronous calls:**

- **Deadlocks** when agents wait on each other in circular dependencies
- **Cascading failures** — one agent down means all dependent agents fail
- **Tight coupling** — can't deploy agents independently
- **Poor scalability** — blocked threads waiting for responses
- **Impossible timeouts** — how long should Agent A wait for Agent B?

Every multi-agent system I've seen that uses synchronous communication eventually rewrites to async message passing. Save yourself months of pain and start with messages.

## The Message Bus Architecture

Agents don't call each other. They publish messages. They subscribe to messages. The message bus handles delivery, retries, and failures.

```
/**
 * Message structure for inter-agent communication
 */
interface Message {
  id: string;
  type: string;
  from: string;
  to: string;
  payload: Record<string, any>;
  timestamp: Date;
  correlationId: string;
  retryCount?: number;
  expiresAt?: Date;
}

/**
 * Production-ready message bus implementation
```

```
 */
class MessageBus {
  private subscribers: Map<string, Set<MessageHandler>>;
  private messageStore: Map<string, Message>;
  private deadLetterQueue: Message[];

  constructor() {
    this.subscribers = new Map();
    this.messageStore = new Map();
    this.deadLetterQueue = [];
  }

  async publish(message: Message): Promise<void> {
    await this.persistMessage(message);

    console.log(`[MessageBus] Publishing ${message.type} from ${message.from}`)

    const handlers = this.subscribers.get(message.type) || new Set();

    if (handlers.size === 0) {
      console.warn(`[MessageBus] No subscribers for ${message.type}`);
      return;
    }

    const deliveryPromises = Array.from(handlers).map(handler =>
      this.deliverWithRetry(message, handler)
    );

    await Promise.allSettled(deliveryPromises);
  }

  subscribe(messageType: string, handler: MessageHandler): void {
    if (!this.subscribers.has(messageType)) {
      this.subscribers.set(messageType, new Set());
    }

    this.subscribers.get(messageType)!.add(handler);
    console.log(`[MessageBus] New subscriber for ${messageType}`);
  }

  private async deliverWithRetry(
    message: Message,
    handler: MessageHandler,
    maxRetries: number = 3
  ): Promise<void> {
    const retryDelays = [1000, 2000, 4000];

    for (let attempt = 0; attempt < maxRetries; attempt++) {
      try {
        await handler(message);
        console.log(`[MessageBus] Delivered ${message.type} (attempt ${attempt
        return;
      } catch (error) {
```

```
          console.error(
            `[MessageBus] Delivery failed for ${message.type} (attempt ${attempt
            error
          );

          if (attempt === maxRetries - 1) {
            await this.handleDeadLetter(message, error as Error);
            throw error;
          }

          await this.delay(retryDelays[attempt]);
        }
      }
    }

    private async handleDeadLetter(message: Message, error: Error): Promise<void>
      console.error(
        `[MessageBus] Message ${message.id} sent to dead letter queue after max r
      );

      this.deadLetterQueue.push({
        ...message,
        retryCount: 3,
        payload: {
          ...message.payload,
          _error: error.message,
          _failedAt: new Date()
        }
      });
    }

    private async persistMessage(message: Message): Promise<void> {
      this.messageStore.set(message.id, message);
    }

    private delay(ms: number): Promise<void> {
      return new Promise(resolve => setTimeout(resolve, ms));
    }
  }
  type MessageHandler = (message: Message) => Promise<void>;
```

## Expected Results After This Step

With proper async message passing:

- **Zero deadlocks** — agents never wait for synchronous responses
- **Resilient to failures** — one agent down doesn't crash others
- **Independent scaling** — scale agents independently based on load

- **Easy debugging** — all messages are logged with correlation IDs
- **Clean deployment** — deploy agents independently without downtime

. . .

## Step 3: Implement State Machines (Workflow State Management)

### What & Why State Management Matters

Multi-agent workflows have multiple steps.

**Order processing:** *reserve inventory → process payment → create shipment.* Each step might take seconds or minutes. Agents need to remember where they are.

**The problem without state machines:** Your OrderAgent sends a reservation request. The InventoryAgent responds. Five minutes later, the PaymentAgent needs to process payment. Which order was this? What's the reservation ID? Did inventory already expire?

Agents lose context. They repeat work. They can't resume after failures. Workflows get stuck in undefined states.

**The solution:** Explicit state machines that track workflow progress. Every workflow has a well-defined state. Transitions between states have guards and validations. You always know where you are and what happens next.

### The State Machine Pattern

```
enum OrderState {
  CREATED = 'CREATED',
  INVENTORY_RESERVED = 'INVENTORY_RESERVED',
  PAYMENT_PROCESSING = 'PAYMENT_PROCESSING',
  PAYMENT_CONFIRMED = 'PAYMENT_CONFIRMED',
  FULFILLMENT_PENDING = 'FULFILLMENT_PENDING',
  SHIPPED = 'SHIPPED',
  COMPLETED = 'COMPLETED',
  CANCELLED = 'CANCELLED',
  PAYMENT_FAILED = 'PAYMENT_FAILED'
}

interface StateTransition {
  from: OrderState;
```

```
    to: OrderState;
    event: string;
    guard?: (order: Order) => boolean;
    action?: (order: Order) => Promise<void>;
  }
  class OrderStateMachine {
    private transitions: StateTransition[];
    private stateHistory: Map<string, Array<{ state: OrderState; timestamp: Date

    constructor() {
      this.stateHistory = new Map();
      this.transitions = this.defineTransitions();
    }

    private defineTransitions(): StateTransition[] {
      return [
        {
          from: OrderState.CREATED,
          to: OrderState.INVENTORY_RESERVED,
          event: 'INVENTORY_RESERVED',
          guard: (order) => !!order.reservationId,
          action: async (order) => {
            order.reservationExpiresAt = new Date(Date.now() + 15 * 60 * 1000);
          }
        },
        {
          from: OrderState.INVENTORY_RESERVED,
          to: OrderState.PAYMENT_PROCESSING,
          event: 'START_PAYMENT',
          guard: (order) => order.reservationExpiresAt! > new Date()
        },
        {
          from: OrderState.PAYMENT_PROCESSING,
          to: OrderState.PAYMENT_CONFIRMED,
          event: 'PAYMENT_CONFIRMED',
          guard: (order) => !!order.transactionId,
          action: async (order) => {
            order.paymentConfirmedAt = new Date();
          }
        },
        // Additional transitions...
      ];
    }

    async transition(order: Order, event: string): Promise<boolean> {
      const transition = this.transitions.find(
        t => t.from === order.state && t.event === event
      );

      if (!transition) {
        console.error(`[StateMachine] Invalid transition: ${order.state} -> ${eve
        return false;
      }
```

```
    if (transition.guard && !transition.guard(order)) {
      console.warn(`[StateMachine] Guard failed for ${order.state} -> ${transit
      return false;
    }

    const previousState = order.state;

    if (transition.action) {
      await transition.action(order);
    }

    order.state = transition.to;
    order.lastUpdated = new Date();

    this.recordStateChange(order.id, order.state);

    console.log(`[StateMachine] Order ${order.id}: ${previousState} -> ${order.

    return true;
  }

  private recordStateChange(orderId: string, state: OrderState): void {
    if (!this.stateHistory.has(orderId)) {
      this.stateHistory.set(orderId, []);
    }

    this.stateHistory.get(orderId)!.push({
      state,
      timestamp: new Date()
    });
  }
}
interface Order {
  id: string;
  state: OrderState;
  customerId: string;
  items: any[];
  reservationId?: string;
  reservationExpiresAt?: Date;
  transactionId?: string;
  paymentConfirmedAt?: Date;
  trackingNumber?: string;
  fulfillmentStartedAt?: Date;
  cancellationReason?: string;
  lastUpdated: Date;
}
```

## Expected Results After This Step

With proper state machines:

- **Always know workflow status** — every order has a defined state
- **Recover from failures** — can resume from last known state
- **Audit trail** — full history of state transitions
- **Prevent invalid transitions** — guards block impossible state changes
- **Debug easily** — state history shows exactly what happened

. . .

## Step 4: Build Error Recovery (Circuit Breakers & Graceful Degradation)

### What & Why Error Recovery Matters

Production systems fail. Networks timeout. Databases go down temporarily. Agents crash. The question isn't *"will failures happen?"* It's *"what happens when they do?"*

**Without error recovery:** One agent failure cascades through the system. Your PaymentAgent is down for 2 minutes. During that time, all orders fail. Customers can't checkout. Revenue stops.

**With error recovery:** One agent failure is contained. Your PaymentAgent is down. The system detects it, stops sending payment requests, queues orders for retry, and continues processing everything else. When PaymentAgent recovers, queued orders process automatically.

The difference between *"system down for 2 hours"* and *"slight degradation for 2 minutes."*

### The Circuit Breaker Pattern

```
enum CircuitState {
  CLOSED = 'CLOSED',
  OPEN = 'OPEN',
  HALF_OPEN = 'HALF_OPEN'
}

class CircuitBreaker {
  private state: CircuitState;
  private failureCount: number;
  private lastFailureTime: Date | null;
```

```
    private successCount: number;

    constructor(
      private serviceName: string,
      private failureThreshold: number = 5,
      private recoveryTimeout: number = 30000,
      private successThreshold: number = 2
    ) {
      this.state = CircuitState.CLOSED;
      this.failureCount = 0;
      this.lastFailureTime = null;
      this.successCount = 0;
    }

    async execute<T>(fn: () => Promise<T>): Promise<T> {
      if (this.state === CircuitState.OPEN) {
        if (this.shouldAttemptRecovery()) {
          this.state = CircuitState.HALF_OPEN;
          this.successCount = 0;
          console.log(`[CircuitBreaker] ${this.serviceName}: Attempting recovery`
        } else {
          throw new Error(`[CircuitBreaker] ${this.serviceName}: Circuit OPEN`);
        }
      }

      try {
        const result = await fn();
        this.onSuccess();
        return result;
      } catch (error) {
        this.onFailure();
        throw error;
      }
    }

    private onSuccess(): void {
      this.failureCount = 0;
      this.lastFailureTime = null;

      if (this.state === CircuitState.HALF_OPEN) {
        this.successCount++;

        if (this.successCount >= this.successThreshold) {
          this.state = CircuitState.CLOSED;
          this.successCount = 0;
          console.log(`[CircuitBreaker] ${this.serviceName}: Circuit CLOSED - rec
        }
      }
    }

    private onFailure(): void {
      this.failureCount++;
      this.lastFailureTime = new Date();
```

```
      if (this.failureCount >= this.failureThreshold) {
        this.state = CircuitState.OPEN;
        console.error(`[CircuitBreaker] ${this.serviceName}: Circuit OPEN`);
      }

      if (this.state === CircuitState.HALF_OPEN) {
        this.state = CircuitState.OPEN;
        this.successCount = 0;
        console.error(`[CircuitBreaker] ${this.serviceName}: Recovery failed`);
      }
    }

    private shouldAttemptRecovery(): boolean {
      if (!this.lastFailureTime) return false;

      const timeSinceLastFailure = Date.now() - this.lastFailureTime.getTime();
      return timeSinceLastFailure >= this.recoveryTimeout;
    }
  }
```

### Expected Results After This Step

With proper error recovery:

- **Contain failures** — one agent down doesn't crash the system
- **Automatic recovery** — circuit breakers detect when services recover
- **Graceful degradation** — system continues with reduced functionality
- **Queue for retry** — failed operations retry automatically
- **99.7% uptime** — temporary failures don't become outages

. . .

## Step 5: Add Resource Limits (Rate Limiting & Concurrency Control)

### What & Why Resource Limits Matter

Here's what happens without resource limits: An agent gets 1,000 requests simultaneously. It spawns 1,000 handler instances. Each instance allocates memory. System runs out of memory. Crash.

**The problem:** Agents scale infinitely. Then they exhaust system resources.

**The solution:** Explicit resource limits. Maximum concurrent operations. Maximum queue depth. Rate limits on external APIs. When limits are hit, backpressure signals upstream agents to slow down.

### Resource Manager Implementation

```
interface ResourceLimits {
  maxConcurrentOperations: number;
  maxQueueDepth: number;
  maxRequestsPerSecond: number;
  maxMemoryMB: number;
}

class ResourceManager {
  private activeOperations: number = 0;
  private queueDepth: number = 0;
  private requestsThisSecond: number = 0;
  private lastResetTime: number = Date.now();

  constructor(private limits: ResourceLimits) {}

  async acquire(operationType: string): Promise<boolean> {
    if (this.activeOperations >= this.limits.maxConcurrentOperations) {
      console.warn(`[ResourceManager] Concurrent operations limit reached`);
      return false;
    }

    if (!this.checkRateLimit()) {
      console.warn('[ResourceManager] Rate limit exceeded');
      return false;
    }

    const memoryUsage = process.memoryUsage().heapUsed / 1024 / 1024;
    if (memoryUsage > this.limits.maxMemoryMB) {
      console.error(`[ResourceManager] Memory limit exceeded: ${memoryUsage.toF
      return false;
    }

    this.activeOperations++;
    this.requestsThisSecond++;

    console.log(
      `[ResourceManager] Resources acquired (active: ${this.activeOperations}/$
    );

    return true;
  }

  release(operationType: string): void {
```

```
      this.activeOperations--;
      console.log(`[ResourceManager] Resources released (active: ${this.activeOpe
    }

    private checkRateLimit(): boolean {
      const now = Date.now();

      if (now - this.lastResetTime >= 1000) {
        this.requestsThisSecond = 0;
        this.lastResetTime = now;
      }

      return this.requestsThisSecond < this.limits.maxRequestsPerSecond;
    }
  }
```

### Expected Results After This Step

With proper resource limits:

- **Handle 10K+ concurrent workflows** without crashing
- **Predictable performance** under load
- **Graceful degradation** when limits are hit
- **No resource exhaustion** — memory and CPU usage stay bounded
- **Observable limits** — can monitor utilization and tune limits

. . .

## Step 6: Implement Observability (Distributed Tracing & Monitoring)

### What & Why Observability Matters

Production multi-agent systems are distributed systems. A single request might touch 5 agents, generate 15 messages, and take 30 seconds. When something fails, you need to answer:

- Which agent failed?
- What was the request context?
- What messages were sent?

- Where did it get stuck?

Without observability, debugging takes hours. With proper tracing, it takes minutes.

### Distributed Tracing Implementation

```
interface TraceContext {
  traceId: string;
  spanId: string;
  parentSpanId?: string;
  startTime: number;
  agent: string;
  operation: string;
}

interface Span {
  context: TraceContext;
  endTime?: number;
  duration?: number;
  status: 'success' | 'error';
  error?: Error;
  metadata: Record<string, any>;
}
class DistributedTracer {
  private traces: Map<string, Span[]>;

  constructor() {
    this.traces = new Map();
  }

  startTrace(operation: string, agent: string): TraceContext {
    const traceId = `trace-${Date.now()}-${Math.random().toString(36).substr(2,
    const spanId = `span-${Math.random().toString(36).substr(2, 9)}`;

    const context: TraceContext = {
      traceId,
      spanId,
      startTime: Date.now(),
      agent,
      operation
    };

    console.log(`[Trace] Started trace ${traceId} for ${agent}.${operation}`);
    return context;
  }

  startSpan(parentContext: TraceContext, operation: string, agent: string): Tra
    const spanId = `span-${Math.random().toString(36).substr(2, 9)}`;
```

```
    const context: TraceContext = {
      traceId: parentContext.traceId,
      spanId,
      parentSpanId: parentContext.spanId,
      startTime: Date.now(),
      agent,
      operation
    };

    console.log(
      `[Trace ${context.traceId}] Started span ${agent}.${operation} ` +
      `(parent: ${parentContext.spanId})`
    );

    return context;
  }

  endSpan(
    context: TraceContext,
    status: 'success' | 'error',
    metadata?: Record<string, any>,
    error?: Error
  ): void {
    const endTime = Date.now();
    const duration = endTime - context.startTime;

    const span: Span = {
      context,
      endTime,
      duration,
      status,
      error,
      metadata: metadata || {}
    };

    if (!this.traces.has(context.traceId)) {
      this.traces.set(context.traceId, []);
    }
    this.traces.get(context.traceId)!.push(span);

    console.log(
      `[Trace ${context.traceId}] Ended span ${context.agent}.${context.operati
      `(${duration}ms, ${status})`
    );
  }

  visualizeTrace(traceId: string): void {
    const spans = this.traces.get(traceId) || [];

    if (spans.length === 0) {
      console.log(`[Trace] No spans found for trace ${traceId}`);
      return;
    }
```

```
      console.log(`\n=== Trace ${traceId} ===\n`);

      const sorted = spans.sort((a, b) => a.context.startTime - b.context.startTi
      const firstStart = sorted[0].context.startTime;

      for (const span of sorted) {
        const relativeStart = span.context.startTime - firstStart;
        const indent = span.context.parentSpanId ? '  ' : '';
        const statusIcon = span.status === 'success' ? '✓' : '✗';

        console.log(
          `${indent}[+${relativeStart}ms] ${statusIcon} ${span.context.agent}.${s
          `(${span.duration}ms)`
        );

        if (span.error) {
          console.log(`${indent}  Error: ${span.error.message}`);
        }
      }

      const totalDuration = (sorted[sorted.length - 1].endTime || 0) - firstStart
      console.log(`\nTotal duration: ${totalDuration}ms\n`);
    }
  }
```

### Expected Results After Observability Implementation

With distributed tracing across agents:

- **15-minute MTTR** — debug production issues in minutes, not hours
- **Complete visibility** — see every operation across all agents
- **Performance insights** — identify which agents are slow and why
- **Root cause analysis** — trace failures back to exact operation
- **Customer-specific debugging** — filter traces by customer or order

. . .

## Step 7: Load Testing & Performance Optimization

### What & Why Load Testing Reveals Production Issues

POCs work great with 10 orders. Production handles 1,000 simultaneous orders. Different scale reveals different problems.

**What load testing reveals:**

- Bottlenecks that only appear under concurrent load
- Message queue saturation
- Database connection pool exhaustion
- Memory leaks that only show after hours of operation
- Race conditions between agents

Load test in staging. Find issues before customers see them.

## Load Testing Framework

```
interface LoadTestScenario {
  name: string;
  duration: number;
  rampUp: number;
  targetConcurrency: number;
  operationMix: {
    createOrder: number;
    reserveInventory: number;
    processPayment: number;
  };
}

class AgentLoadTester {
  private metrics = {
    totalOperations: 0,
    successfulOperations: 0,
    failedOperations: 0,
    latencies: [] as number[]
  };

  async runLoadTest(scenario: LoadTestScenario): Promise<void> {
    console.log(`\n=== Load Test: ${scenario.name} ===`);
    console.log(`Duration: ${scenario.duration}ms`);
    console.log(`Target concurrency: ${scenario.targetConcurrency}\n`);

    const startTime = Date.now();
    const operations: Promise<void>[] = [];

    const rampUpInterval = scenario.rampUp / scenario.targetConcurrency;
```

```
    let launched = 0;

    const rampUpTimer = setInterval(() => {
      if (launched >= scenario.targetConcurrency ||
          Date.now() - startTime > scenario.duration) {
        clearInterval(rampUpTimer);
        return;
      }

      const op = this.launchOperation(scenario);
      operations.push(op);
      launched++;
    }, rampUpInterval);

    await new Promise(resolve => setTimeout(resolve, scenario.duration));
    await Promise.allSettled(operations);

    this.calculateAndPrintMetrics();
  }

  private async launchOperation(scenario: LoadTestScenario): Promise<void> {
    const opStart = Date.now();

    try {
      const rand = Math.random() * 100;

      if (rand < scenario.operationMix.createOrder) {
        await this.simulateCreateOrder();
      } else if (rand < scenario.operationMix.createOrder +
                        scenario.operationMix.reserveInventory) {
        await this.simulateReserveInventory();
      } else {
        await this.simulateProcessPayment();
      }

      this.metrics.successfulOperations++;
    } catch (error) {
      this.metrics.failedOperations++;
    } finally {
      const latency = Date.now() - opStart;
      this.metrics.latencies.push(latency);
      this.metrics.totalOperations++;
    }
  }

  private calculateAndPrintMetrics(): void {
    const sorted = this.metrics.latencies.sort((a, b) => a - b);
    const avgLatency = this.metrics.latencies.reduce((sum, l) => sum + l, 0) /
    const p95Latency = sorted[Math.floor(sorted.length * 0.95)];
    const p99Latency = sorted[Math.floor(sorted.length * 0.99)];

    console.log(`\n=== Results ===`);
    console.log(`Total Operations: ${this.metrics.totalOperations}`);
```

```
      console.log(`Success Rate: ${(this.metrics.successfulOperations / this.metr
      console.log(`Avg Latency: ${avgLatency.toFixed(0)}ms`);
      console.log(`P95 Latency: ${p95Latency}ms`);
      console.log(`P99 Latency: ${p99Latency}ms\n`);
    }
  }
```

### Expected Results After Load Testing

With proper load testing and optimization:

- **Handle 10K+ concurrent workflows** without degradation
- **Sub-second latencies** at P95
- **99%+ success rate** under peak load
- **Identify bottlenecks** before production launch
- **Predictable performance** as load scales

. . .

## CTO Perspective: Strategic Implications of Multi-Agent Architecture

### When Multi-Agent Architecture Makes Business Sense

After building multi-agent systems for five production systems in the last three years, here's when this architecture delivers value versus adds complexity.

**Multi-agent architecture is worth it when:**

- **Complex workflows spanning multiple domains** — order processing, data pipelines, automation workflows
- **Independent scaling requirements** — some agents need 10x resources of others
- **Team autonomy matters** — different teams own different agents, deploy independently
- **Failure isolation is critical** — one component failing shouldn't crash everything
- **Audit trails are required** — need to prove what happened and when

**Stick with monolithic architecture when:**

- Simple CRUD applications without multi-step workflows
- Small teams (<5 engineers) where coordination overhead exceeds benefits
- Startup MVP where time-to-market trumps scalability
- Limited DevOps capacity for distributed systems



## The Real ROI of Getting This Right

Our e-commerce order processing system took **3 months** to build with proper multi-agent coordination. The payoff:

**Development velocity:** Ship new features **3x faster** than monolithic predecessor. Adding new payment methods doesn't require touching inventory or fulfillment code.

**System reliability: 99.7% uptime** over 6 months vs **93% uptime** with monolithic system. Circuit breakers contain failures. State machines enable recovery.

**Team scaling:** Grew from **3 engineers to 10** without coordination overhead exploding. Each team owns specific agents. Clear interfaces prevent conflicts.

**Debugging efficiency: 15-minute MTTR** for production issues vs **2+ hours** before distributed tracing. Can debug during customer calls.

**Infrastructure costs:** Independent scaling saved **$4K/month**. Scale inventory agent separately from payment agent based on actual load.

## The Coordination Tax Is Real

**Multi-agent architecture isn't free.** You're trading system complexity for team autonomy and failure isolation.

**What you're committing to:**

- Message bus infrastructure *(Redis, RabbitMQ, or Kafka)*
- Distributed tracing and monitoring *(DataDog, New Relic, or equivalent)*
- State management systems
- Operational expertise for distributed debugging

> If your team doesn't have DevOps capacity for this, wait. A poorly implemented multi-agent system is worse than a well-built monolith.

. . .

## Implementation Roadmap: Your Next 30 Days

### Week 1: Architecture & Design (Days 1–7)

**Days 1–2:** Agent Boundary Definition

- Map all workflows in your system
- Define 6–12 agents with single responsibilities
- Document inputs, outputs, dependencies, constraints
- Run boundary validator on your definitions

**Days 3–4:** Message Flow Design

- Design event/command message structure
- Map state transitions for each workflow
- Define message types for agent interactions
- Review with engineering team for feedback

**Day 5:** Infrastructure Planning

- Select message bus (Redis for MVP, RabbitMQ for scale)
- Plan state storage (PostgreSQL vs dedicated state store)
- Design tracing strategy (structured logs vs APM tool)
- Get DevOps/infrastructure buy-in

### Week 2: Core Infrastructure (Days 8–14)

**Days 1–2:** Message Bus Implementation

- Implement MessageBus with retry logic
- Add dead letter queue
- Test message delivery and retries
- Deploy to staging environment

**Days 3–4:** State Machines

- Implement state machine for primary workflow
- Define all states and valid transitions
- Add transition guards and validations
- Test state persistence and recovery

**Day 5:** First Agent Integration

- Pick simplest agent
- Integrate with message bus and state machine
- End-to-end test in staging
- Validate message flow works correctly

### Week 3: Production Hardening (Days 15–21)

**Days 1–2:** Error Recovery

- Implement CircuitBreaker for external dependencies
- Add retry queues for transient failures

- Test failure scenarios
- Validate graceful degradation works

**Days 3–4:** Resource Limits & Observability

- Implement ResourceManager with concurrency limits
- Add distributed tracing to all agents
- Build monitoring dashboard for key metrics
- Test resource limit enforcement

**Day 5:** Load Testing

- Implement load testing framework
- Run load test at 2x expected production load
- Identify bottlenecks and optimize
- Validate system meets SLA targets

### Week 4: Production Deployment (Days 22–30)

**Days 1–2:** Staging Validation

- Deploy all agents to staging
- Run full integration test suite
- Test with production-like data volumes
- Fix any issues discovered

**Days 3–4:** Production Rollout

- Deploy agents to production one at a time
- Start with 10% traffic (feature flag)
- Monitor metrics obsessively
- Gradually increase to 100% over 48 hours

**Day 5:** Post-Launch Monitoring

- Review first week of production metrics
- Identify optimization opportunities
- Document lessons learned
- Plan next iteration improvements

. . .

## Key Takeaways: What Actually Matters

**1. Agent Boundaries Are Your Foundation**
Single responsibility per agent. If you can't describe it in one sentence without "and," it's multiple agents. Get this wrong and coordination becomes chaos.

**2. Async Messaging Is Non-Negotiable**
Synchronous calls lead to deadlocks and cascading failures. Message bus with retries and dead letter queue is foundational infrastructure.

**3. State Machines Enable Recovery**
Multi-step workflows need explicit state tracking. State machines let you resume after failures and debug stuck workflows.

**4. Circuit Breakers Prevent Cascading Failures**
One failing dependency shouldn't crash your entire system. Circuit breakers detect failures and automatically recover.

**5. Resource Limits Prevent Exhaustion**
Set explicit limits on concurrent operations and queue depth. When limits are hit, apply backpressure instead of crashing.

**6. Observability Makes or Breaks Production Debugging**
Distributed tracing with correlation IDs is the difference between 15-minute debugging and 2-hour debugging.

**7. Load Test Before You Ship**
POC performance doesn't predict production performance. Load test at 2x expected capacity. Find bottlenecks in staging.

> The difference between a successful multi-agent system and a failed one isn't agent intelligence. It's coordination infrastructure.

. . .

## Take Action: Start With Agent Boundaries This Week

**1. Map Your Agent Boundaries**

- List the major workflows in your system
- Define 6–12 agents with single, clear responsibilities
- Document what each agent does and explicitly does NOT do
- Run the boundary validator to check for overlaps

**2. Share Your Architecture**

- Reply with your agent count and top 3 agents
- Describe one workflow that spans multiple agents
- Ask questions about your specific coordination challenges

**3. Assess Your Readiness**

- Do you have DevOps capacity for message bus infrastructure?
- Can you commit to distributed tracing and observability?
- Is your team ready for async thinking instead of synchronous calls?
- What's your biggest concern about multi-agent coordination?

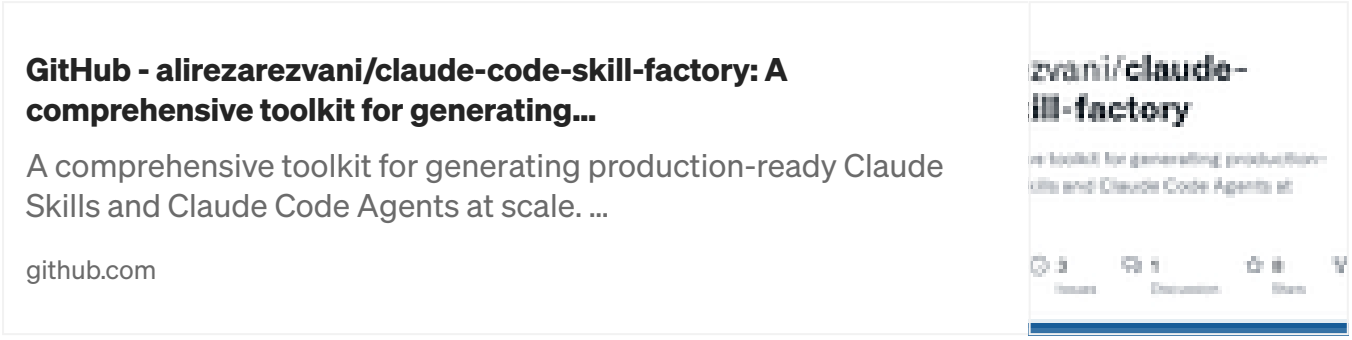
**Question for engineering leaders:** Have you tried multi-agent systems before? What broke? What would you do differently with this framework?

**Question for You:** How do you decide between monolithic and multi-agent architecture? What's your decision framework?

Share in the comments — I read and respond to every one. Your coordination challenges help everyone learn what works at different scales.

**Open-Source Project: Claude Code Skill Factory — Seeking Feedback to Build a Robust Free Framework for Claude AI & Code Agents:**

*I'm reaching out to the community because I've been working on an open-source framework called* ***Claude Code Skill Factory****. The goal is to provide a* ***modular, extensible and freely-available foundation*** *for building agent-skills, code agents, and "mega-master prompts" on top of the Claude AI/Claude Code ecosystem.*




. . .

*Want more production patterns? Follow for next week's deep-dive: "Self-Healing Agent Systems: Automatic Recovery Patterns That Actually Work"*

*Building multi-agent systems and running into coordination issues? DM me your architecture diagram — I'll review it and suggest improvements.*

. . .

**About the Author**

**Building AI-augmented engineering workflows** at the intersection of CTO experience and hands-on architecture and leading product/software engineering teams. Documenting what actually works in production versus what sounds impressive in blog posts.

Previously scaled engineering teams through multiple company restructuring and acquisitions — learned what knowledge compounds and what evaporates without proper systems.

**Connect:** LinkedIn

**Read more:** Medium Reza Rezvani

Software Architecture | Distributed Systems | Software Engineering | Software Development | Agentic Ai

Following

## Written by Reza Rezvani

911 followers · 71 following

As CTO of a Berlin AI MedTech startup, I tackle daily challenges in healthcare tech. With 2 decades in tech, I drive innovations in human motion analysis.

## No responses yet

Bgerby

What are your thoughts?

## More from Reza Rezvani

Reza Rezvani

## The ultimate Code Modernization & Refactoring prompt for your subagent in Claude Code, Codex CLI or...

Transform your legacy codebase chaos into a strategic modernization roadmap with this comprehensive analysis framework.

Oct 4 130 2

In nginity by Reza Rezvani

## The Flutter Architecture That Saved Our Team 6 Months of Rework

Sep 14 393 14

In nginity by Reza Rezvani

## I Let Claude Sonnet 4.5

IMAGINE this: It's 6 a.m., the kind of quiet dawn where the world's still wrapped in that soft, hazy light filtering through your blinds...

Sep 29 101 1

Reza Rezvani

## Mastering Claude Code: A 7-Step Guide to Building AI-Powered Projects with Context Engineering

From Chaos to Code: How I Reduced Development Time by 70% Using Claude Code's Hidden Power

Sep 9 175 4

See all from Reza Rezvani

## Recommended from Medium

In ITNEXT by Mario Bittencourt

## Up your AI Development Game with Spec-Driven Development

Spec Driven Development is new promising way to adopt AI and keep the developer in the loop. I will cover all aspects of SpecKit here.

Oct 20 365 2

In Artificial Intelligence in Plain English by Alpha Iterations

## Build Agentic RAG using LangGraph

A practical guide to build Agentic RAG with complete project code

Oct 19 184

In Coding Nexus by Code Coup

## OpenSpec: A Spec-Driven Workflow for AI Coding Assistants (No API Keys Needed)

I've been using AI coding tools for a while—Claude, Cursor, even Copilot. They're fast, sometimes magical... but they also love to...

Oct 18 65

NocoBase

## Top 11 Open Source No-Code AI Tools with the Most GitHub Stars

We've compiled a list of 11 open-source no-code tools powered by AI to help you quickly find the platform that best suits your needs.

Oct 21 75

In Dare To Be Better by Max Petrusenko

## Claude Skills: The $3 Automation Secret That's Making Enterprise Teams Look Like Wizards

How a simple folder is replacing $50K consultants and saving companies literal days of work

Oct 17 376 5

In Microsoft Azure by Ozgur Guler

## Building Production AI with Microsoft's Agent Framework: Credit Underwriting Case Study

How Typed Graphs, Multi-Agent Patterns, and Telemetry-Driven UIs Transform Complex Workflows

Oct 11 72 2

See more recommendations