The Code Modernization Playbook

Transforming Legacy Systems with AI

Claude

Contents

Modernize – or get left behind

IT and engineering organizations are at a crossroads.

Maintain legacy systems or upgrade them. Keep the lights on for your COBOL-powered mainframe or allocate precious engineering resources to migrating your codebase to Python or Java. Stick with your battle-tested monolith or embark on the complex journey of decomposing it into microservices.

With a never-ending list of new features to build and stakeholders to appease, it can be nearly impossible to find the time or resources to future-proof your codebase. But the decision not to re-architect your stack goes beyond the maintenance cost of technical debt—it represents a fundamental threat to your company's competitive advantage.

You know code modernization is the way forward, but all too often the short-term cost of delaying more pressing projects is too steep to pay.

The good news? You don't have to choose between code modernization and shipping your next great feature. Enter: agentic code modernization.

The case for agentic code modernization

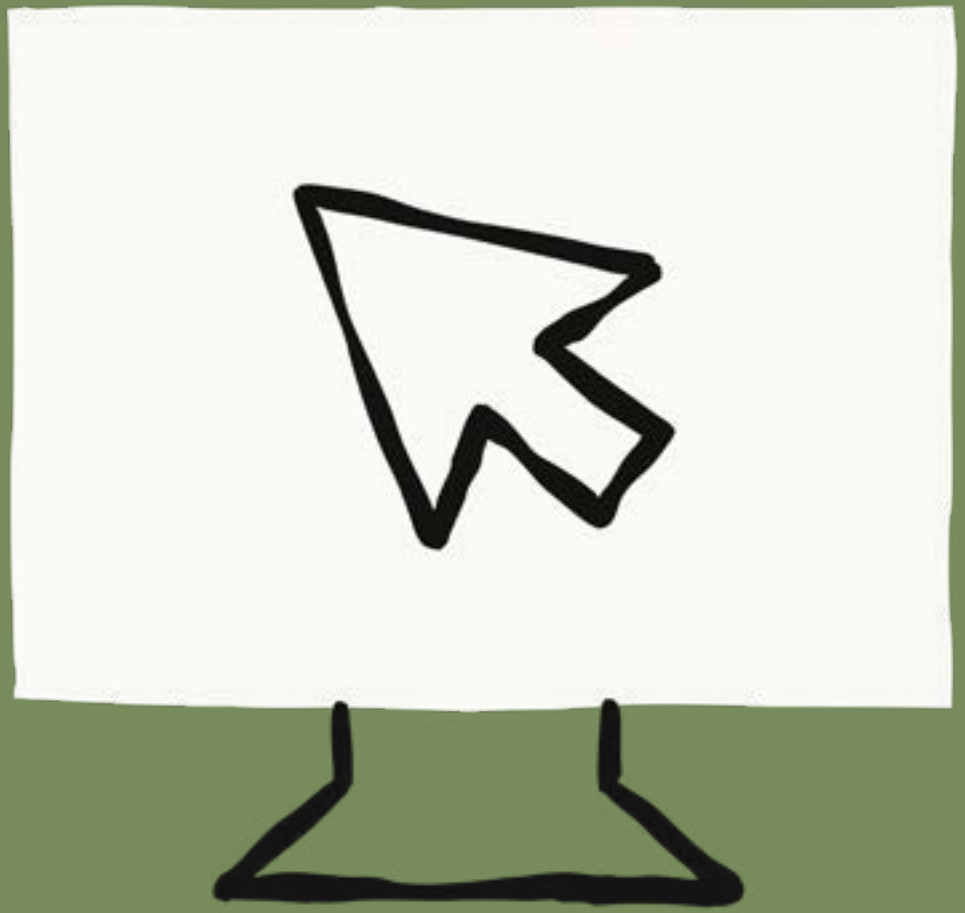
Agentic code modernization transcends simple language migration; it represents a fundamental transformation of how organizations build, maintain and evolve their software systems.

Recent advances in agentic coding tools enable development teams to tackle modernization projects that were previously deemed too complex, risky, or resource-intensive. These solutions embed powerful coding models directly into development workflows, providing deep codebase awareness and the ability to securely edit files and run commands directly in existing development environments.

The emergence of tools like **Claude Code**, which leverages advanced coding models like **Sonnet 4** and **Opus 4.1**, marks a paradigm shift in how organizations approach legacy system transformation. These tools understand context across entire codebases, preserve critical business logic during migrations and generate comprehensive documentation where none previously existed.

This playbook delivers actionable strategies for identifying high-ROI modernization opportunities that align with strategic business objectives. Technical leaders will discover how to build compelling business cases that secure executive buy-in and funding by quantifying both the costs of inaction and the value of transformation.

Let's dive in.



Chapter 1

Three trends blocking digital transformation

Chapter 1

Three trends blocking digital transformation

From cloud computing to generative AI, digital transformation has become a critical imperative for organizations seeking to remain competitive in today's rapidly evolving technological landscape. However, despite significant investments in the tools, platforms and talent to support these initiatives, many IT and engineering organizations find themselves struggling to achieve tangible ROI.

In this chapter, we examine three critical trends blocking digital transformation–and what leaders can do about it.

Decreasing developer productivity

According to McKinsey, the average developer spends **17.3 hours each week** dealing with technical debt, bad code and maintenance tasks like debugging and refactoring instead of building. This maintenance burden creates a vicious cycle where innovation becomes nearly impossible, as teams find themselves constantly fighting fires rather than shipping new features. Developer burnout accelerates as talented engineers lose motivation working on outdated systems that offer little opportunity for professional growth or creative problem-solving. The psychological impact extends beyond individual contributors to entire teams, creating a culture of resignation where "that's just how things work here" becomes the default response to systemic inefficiencies.

Talent acquisition and retention

As software technology evolves at an exponential pace, expertise in older systems becomes increasingly scarce, with fewer technical professionals possessing proficiency in legacy programming languages and architectures. Meanwhile, young developers graduating from computer science programs have been trained on modern languages and cloud-native architectures, making positions that require COBOL, Fortran or even older versions of Java increasingly difficult to fill.

Despite the fact that COBOL is often considered a "dying" language, it was **estimated by the COBOL Working Group** of the Open Mainframe Project in 2021 that there are 250 billion lines of COBOL in use at businesses worldwide. Organizations find themselves competing for a shrinking pool of expensive specialists to maintain these systems while simultaneously struggling to attract new talent who view legacy system work as career-limiting rather than career-enhancing.

Rising technical debt and security risks

According to a recent survey conducted by **Protiviti**, nearly 70% of organizations cite technical debt as a primary inhibitor on their ability to innovate. In specific industries, the numbers are much more stark, with 82% of biotechnology and 78% of financial services organizations citing technical debt as a blocker to new feature development.

Security vulnerabilities multiply as legacy systems no longer receive patches or updates from vendors who have long since moved on to newer technologies. Each unpatched vulnerability represents a potential entry point for malicious actors, with legacy systems often lacking the monitoring and security controls that modern architectures provide by default. Additionally, compliance requirements continue to evolve while legacy systems remain static, creating an ever-widening gap between what regulations demand and what systems can deliver.

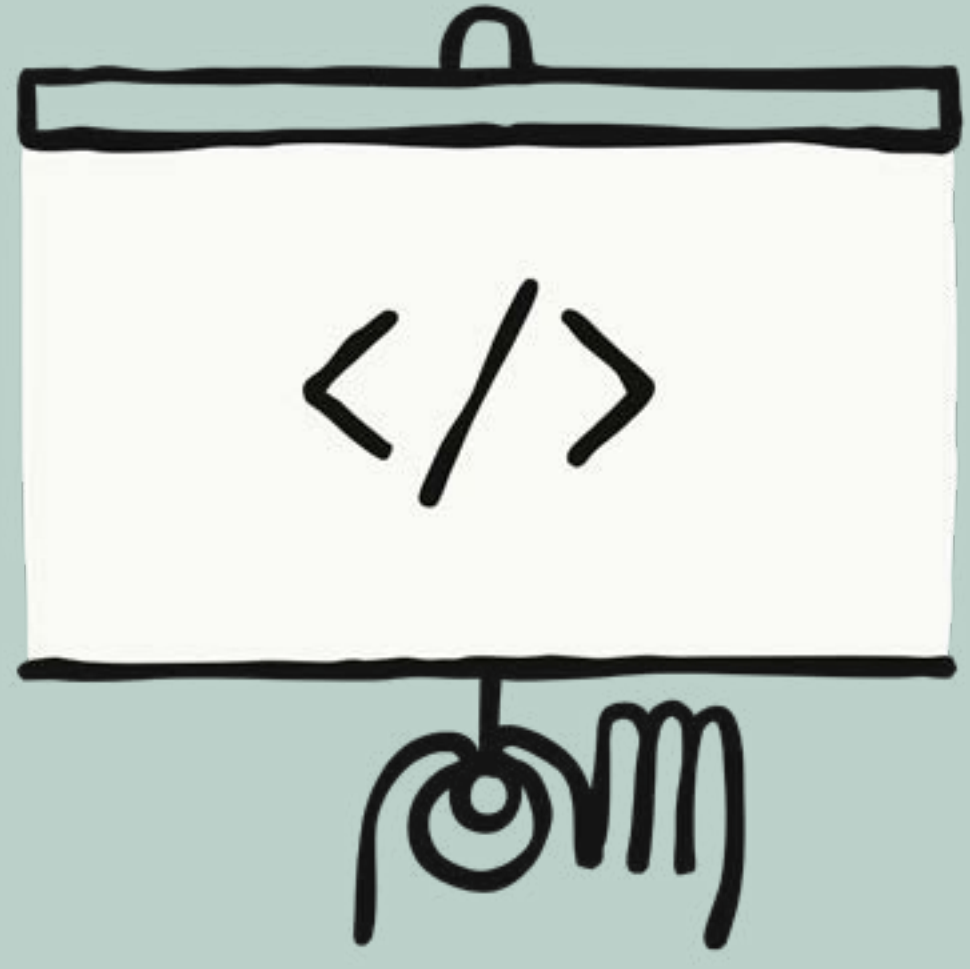
As a result of this debt, financial services organizations struggle to implement real-time fraud detection on batch-processing systems, healthcare providers cannot meet interoperability mandates with siloed databases and retailers face customer data protection requirements that legacy architectures were never designed to support. Organizations find themselves trapped between the fear of change and the escalating risk of maintaining the status quo in an environment of increasing cyber threats and regulatory scrutiny.

The root cause: legacy engineering practices

Across industries, decades-old architectural decisions continue to constrain digital transformation initiatives. Documentation is often lost over time through staff turnover and organizational changes, leaving critical business logic trapped in code that few understand and even fewer can modify safely. The original designers and implementers have often moved on, taking with them the contextual knowledge that made these systems comprehensible.

As a result, organizations develop a culture of fear around touching “working” systems, where the definition of “working” often means “we’re afraid to verify if it actually works correctly.” This paralysis manifests in risk-averse behavior where teams know change is necessary but fear the potential consequences of attempting it, leading to a status quo that becomes increasingly untenable.

There has to be a better way.



Chapter 2

The rise of code modernization

Chapter 2

The rise of code modernization

Let's explore what code modernization actually means for the teams living with legacy systems every day. It's not about throwing away decades of refined business logic—that would be wasteful and risky. Instead, it's about thoughtfully transforming how that logic lives and breathes in your infrastructure.

You've probably seen "lift-and-shift" migrations that promise quick wins but really just relocate problems to fancier real estate (and at a non insubstantial cost). Real modernization goes deeper. It asks better questions: How can your architecture evolve? What tools would help your team work better? Which practices are holding you back?

The three pillars of code modernization

When teams successfully modernize their code, they tend to focus on three interconnected changes: how systems are structured, what technologies power them, and how people work with them. Here's what we've learned:

Architecture transformation

Organizations undertaking architecture transformation must move from monolithic, on-prem structures to microservices and cloud-native patterns that provide flexibility and resilience. This architectural evolution enables independent scaling and deployment of system components, allowing teams to update customer-facing features without putting core transaction processing systems at risk. Development teams can work autonomously without the coordination overhead that monolithic systems require, accelerating delivery while reducing the risk of conflicting changes.

System resilience improves dramatically through the isolation of potential failures, where issues in one service no longer cascade throughout the entire system. Modern architectures implement circuit breakers, retry logic and graceful degradation patterns that maintain service availability even when individual components fail. This architectural approach transforms brittle systems that require complete downtime for updates into antifragile systems that grow stronger through controlled failure and continuous improvement.

Technology stack modernization

To achieve code modernization, outdated languages and frameworks must give way to modern alternatives that support current development practices and attract top talent. New technologies bring immediate benefits in terms of performance, security and developer productivity. Modern runtime environments provide performance improvements that come naturally, with garbage collection, just-in-time compilation and hardware optimization that legacy systems cannot match. Security vulnerabilities decrease significantly with actively maintained technologies that receive regular updates and patches.

The ecosystem advantages of modern technology stacks extend far beyond the core language or framework. Rich libraries, comprehensive tooling and active communities accelerate development while reducing the need to build custom solutions for common problems. Organizations transitioning from COBOL to Java gain access to thousands of open-source libraries, while those moving from proprietary systems to Python unlock powerful data science and machine learning capabilities that can transform business operations.

Development practice evolution

Legacy waterfall methodologies must transform into continuous integration

and continuous deployment (CI/CD) pipelines with automated testing that catches issues before they reach production. Deployment frequency increases from monthly or quarterly releases to multiple deployments per day, enabling organizations to respond rapidly to market changes and customer feedback. Quality improves through comprehensive automated testing that validates not just functionality but also performance, security and compliance requirements.

Time-to-market accelerates dramatically with modern practices that eliminate manual handoffs and reduce coordination overhead. Infrastructure as Code (IaC) ensures consistent environments from development through production, while GitOps practices provide auditable, reversible changes to both code and infrastructure. These practices create a foundation for innovation where experimentation becomes safe and failure becomes a learning opportunity rather than a crisis.

Industries ripe for disruption

Several sectors remain heavily dependent on legacy systems, creating opportunities for innovative competitors to capture market share by offering superior digital experiences and operational efficiency.

Financial services

The financial services industry illustrates the acute challenges imposed by legacy system constraints. Many banks continue to rely on massive COBOL codebases processing trillions in daily transactions, yet these systems struggle to support the real-time processing capabilities that modern customers have come to expect. Running on mainframes that often cost millions just to maintain, these legacy architectures limit banks' ability to offer instant payments, real-time fraud detection, and the personalized services that digital-native competitors provide as standard features. The reliance on overnight batch processing creates frustrating delays for customers who experience instant gratification in other aspects of their digital lives.

Healthcare and pharma

The healthcare and pharmaceutical industries face particular challenges with aging drug development and clinical trial infrastructure. At a time when speed to market can influence both patient outcomes and revenue potential measured in billions, many organizations find their legacy systems may be limiting their competitive capabilities. Statistical computing systems running SAS or proprietary languages often lack the flexibility to incorporate modern AI/ML capabilities that could potentially accelerate drug candidate identification or improve trial outcome predictions. Researchers frequently encounter constraints that reflect outdated technical limitations, such as batch processing requirements for genomic data that contemporary systems can handle in real-time.

Retail

The retail industry's technological evolution highlights the tension between legacy infrastructure and modern customer expectations. Many retailers continue operating inventory systems developed decades ago, which can create challenges in today's omnichannel environment where customers expect seamless experiences across online, mobile and physical stores. When point-of-sale systems only update inventory through nightly batch processes, supporting services like buy-online-pickup-in-store becomes problematic, as does providing the real-time inventory visibility that could prevent customers from making unnecessary trips to find out-of-stock items. The influence of Amazon and other digital-native retailers has shaped customer expectations around real-time visibility and channel integration that legacy systems often struggle to meet.

Manufacturing

Proprietary automation code in manufacturing creates vendor lock-in that prevents Industry 4.0 adoption and smart factory initiatives. Legacy systems designed for isolated production lines cannot integrate with modern enterprise planning and optimization tools that could reduce waste and improve efficiency. The inability to collect and analyze real-time production data prevents manufacturers from implementing predictive maintenance that could prevent costly downtime.

Chapter 3

Real-world code modernization use cases

Chapter 3

Real-world code modernization use cases

Code modernization removes the technical barriers that prevent organizations from implementing innovative customer experiences and operational improvements. Here are five real-world use cases teams can apply today with agentic coding tools like Claude Code.

Language migration

Organizations using agentic coding tools can successfully navigate complex language migrations that preserve business logic while gaining modern language benefits. For example:

- Banking institutions can easily migrate from COBOL to Java for transaction processing systems, enabling cloud deployment.
- VB6 applications can transform into C#/.NET implementations that support web interfaces.
- Data processing pipelines can evolve from Perl scripts to Python frameworks that integrate seamlessly with modern data science tools.

Sample Scenario: An insurance company used Claude Code to migrate their COBOL-based claims processing system to Java. The team began by having Claude analyze their existing architecture. Claude created detailed diagrams that revealed hidden dependencies and business logic. The migration preserved all original business rules while adding modern capabilities like dependency injection, transaction management and real-time event streaming.

```
COBOL

* Original COBOL Claims Processing
  IDENTIFICATION DIVISION.
  PROGRAM-ID. PROCESS-CLAIM.

DATA DIVISION.
WORKING-STORAGE SECTION.
01  WS-CLAIM-RECORD.
  05  WS-CLAIM-ID        PIC 9(8).
  05  WS-POLICY-NUMBER   PIC X(10).
  05  WS-CLAIM-AMOUNT    PIC 9(7)V99.
  05  WS-DEDUCTIBLE      PIC 9(5)V99.
  05  WS-COVERAGE-LIMIT  PIC 9(7)V99.
  05  WS-APPROVAL-STATUS PIC X(1).
    88  APPROVED       VALUE 'A'.
    88  DENIED         VALUE 'D'.
    88  PENDING        VALUE 'P'.
```

Translation to Java, facilitated by Claude Code:

```
Java

// Migrated Java implementation with modern patterns
@Service
@Transactional
public class ClaimProcessingService {

  @Data
  @Builder
  public static class ClaimRecord {
    private Long claimId;
    private String policyNumber;
    private BigDecimal claimAmount;
    private BigDecimal deductible;
    private BigDecimal coverageLimit;
    private ClaimStatus approvalStatus;
  }

  public ClaimRecord processClaim(ClaimRecord claim) {
    // Preserved business logic with enhanced error handling

    validateClaim(claim);
    calculatePayout(claim);
    updateClaimStatus(claim);

    // Modern additions: async processing, event publishing
    publishClaimEvent(claim);
    return claimRepository.save(claim);
  }
}
```

Platform modernization

Agentic coding solutions enable teams to transform legacy platforms into modern, scalable architectures. For example:

- Mainframe batch jobs convert to serverless functions that scale automatically based on demand.
- On-premise systems migrate to Kubernetes clusters that provide resilience through container orchestration.

Sample Scenario: A retail chain utilized Claude Code to convert their mainframe inventory batch processing system to AWS Lambda functions. By prompting Claude Code with "Convert the batch job in /inventory/daily_reconciliation to a serverless function written in Python3," they transformed overnight processing into real-time updates. The transformation eliminated the 24-hour delay in inventory updates and enabled real-time stock checks across stores.

```
Job Control Language

//INVRECON JOB (ACCT),'DAILYINVENTORY',CLASS=A,
MSGCLASS=X
//STEP1    EXEC PGM=INVPRC01
//STEPLIB  DD   DSN=PROD.LOADLIB,DISP=SHR
//INVSALES DD   DSN=PROD.DAILY.SALES,DISP=SHR
//INVMAST  DD   DSN=PROD.INVENTORY.MASTER,DISP=OLD
```

Modernization from an IBM mainframe batch processing system to a severless function written in Python3:

```
Python

# Converted AWS Lambda Function - Real-time inventory processing
import json
import boto3
from decimal import Decimal
def lambda_handler(event, context):
  """Process inventory changes in real-time as they occur"""
  try:
    for record in event['Records']:
      transaction = json.loads(record['body'])

      # Real-time inventory update with DynamoDB
      response = inventory_table.update_item(
        Key={
          'store_id': transaction['store_id'],
          'product_id': transaction['product_id']
        },
        UpdateExpression="SET qty_on_hand = qty_on_hand + :qty_
        change",
        ExpressionAttributeValues={
          ':qty_change': Decimal(str(transaction['quantity_
          change']))
        }
      )

      # Real-time reorder alerts (no more waiting for nightly
      batch)
      if response['Attributes']['qty_on_hand'] <=
      response['Attributes']['reorder_point']:
          send_reorder_alert(response['Attributes'])

  except Exception as e:
    logger.error(f"Error processing inventory: {str(e)}")
    raise
```

Architecture transformation

Modern development teams use agentic coding solutions to break up monolithic applications into microservices that can be developed, deployed and scaled independently. This architectural transformation requires careful analysis of existing code to identify service boundaries, shared data dependencies and transaction boundaries.

Sample Scenario: A financial services firm used Claude Code to decompose their monolithic trading system into microservices. Claude Code analyzed millions of lines of code to identify natural service boundaries around order management, risk calculation and settlement processing. The decomposition enabled independent deployment of services, automatic scaling based on load patterns and fault isolation where settlement failures don't impact order placement.

```
Java

// Original Monolithic Trading System
@Component
public class TradingSystemMonolith {
  @Transactional
  public TradeResult executeTrade(TradeRequest request) {
    // Order validation mixed with risk checks
    if (!validateOrder(request)) {
      return TradeResult.rejected("Invalid order");
    }

    // Risk calculation embedded in order flow
    RiskMetrics risk = calculateRisk(request, currentPosition);
    if (risk.getVaR() > getAccountLimit(request.
    getAccountId())) {
      return TradeResult.rejected("Risk limit exceeded");
    }

    // Settlement logic intertwined with order execution
    Order order = new Order(request);
    db.insert("INSERT INTO orders VALUES (?)", order);

    // Synchronous settlement causing bottlenecks
    Settlement settlement = settlementProcessor.process(order);

    return TradeResult.success(order.getId());
  }
}
```

Decomposition from monolith to microservices:

```
Java

// Decomposed Order Management Microservice
@RestController
@RequestMapping("/api/v1/orders")
public class OrderService {

  @PostMapping
  @CircuitBreaker(name = "order-creation")
  public ResponseEntity<OrderResponse> createOrder(@RequestBody
  OrderRequest request) {
    // Focused solely on order management
    Order order = Order.builder()
      .accountId(request.getAccountId())
      .symbol(request.getSymbol())
      .quantity(request.getQuantity())
      .status(OrderStatus.PENDING_RISK_CHECK)
      .build();

    order = orderRepository.save(order);

    // Asynchronous event-driven communication
    eventPublisher.publish(new OrderCreatedEvent(order));

    return ResponseEntity.accepted().body(OrderResponse.
    from(order));
  }
}

// Risk Calculation Microservice
@Service
public class RiskService {
  @EventListener

  @Async
  public void handleOrderCreated(OrderCreatedEvent event) {
    // Independent risk calculation with its own data store
    RiskAssessment assessment = performRiskAssessment(event.
    getOrder());

    if (assessment.isApproved()) {
      publishEvent(new RiskApprovedEvent(event.getOrderId(),
      assessment));
    } else {
      publishEvent(new RiskRejectedEvent(event.getOrderId(),
      assessment));
    }
  }
}
```

Integration modernization

Legacy integration patterns create significant maintenance burdens. Point-to-point integrations evolve into complex webs of dependencies, while file transfer protocols struggle to meet real-time requirements. Modern integration architectures consolidate these into managed API gateways that provide centralized security, monitoring and version management.

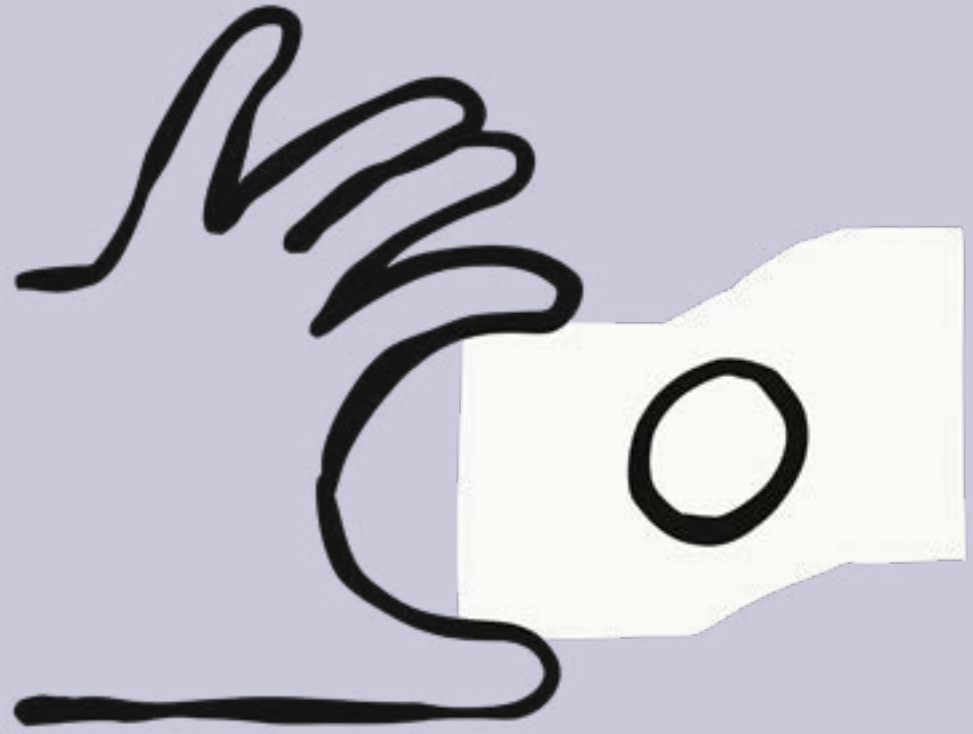
Sample Scenario: A logistics company transformed their FTP-based partner integration system to a modern REST API platform using Claude Code. Through prompts like “Convert this FTP-based file exchange protocol to a REST API design that supports real-time updates,” they reduced integration errors by 90%. The transformation delivered a reduction in integration errors through structured validation, real-time tracking via WebSocket connections instead of FTP delays, and backward compatibility maintained with CSV batch upload endpoint.

```
Bash

# Original FTP-based Integration
#!/bin/bash
# Partners upload CSV files to FTP every 4 hours
ftp -inv $FTP_SERVER << EOF
user $FTP_USER $FTP_PASS
cd /inbound
mget PARTNER_SHIPMENT_*.csv
bye
EOF

# Process each file sequentially
for file in PARTNER_SHIPMENT_*.csv; do
  ./process_shipment.pl $file  # No error handling
  mv $file ./processed/
done
```

Transformation from legacy integration system written in Bash to REST API design written in Typescript:

Typescript

```typescript
// Modern REST API Implementation
export class ShipmentAPIRouter {
  private setupRoutes() {
    // Real-time shipment creation with validation
    this.router.post('/api/v2/shipments',
      this.authenticate,
      this.validateRequest(ShipmentSchema),
      this.rateLimit,
      async (req: Request, res: Response) => {
        try {
          const shipment = await this.createShipment(req.body,
          req.partner);

          res.status(201).json({
            id: shipment.id,
            status: 'CREATED',
            trackingUrl: `https://track.logistics.com/${shipment.
            id}`,
            _links: {
              self: `/api/v2/shipments/${shipment.id}`,
              events: `/api/v2/shipments/${shipment.id}/events`
            }
          });

          // Real-time notifications
          this.eventBus.broadcast(shipment.id, {
            type: 'SHIPMENT_CREATED',
            timestamp: new Date().toISOString(),
            data: shipment
          });
        } catch (error) {
          this.handleError(error, res);
        }
      }
    );

    // Backward compatibility endpoint
    this.router.post('/api/v2/shipments/batch',
      this.authenticate,
      upload.single('file'),
      async (req: Request, res: Response) => {
        // Support legacy CSV format while providing modern API
        benefits
        const shipments = req.body.format === 'csv'
          ? await this.parseCSVFile(req.file)
          : JSON.parse(req.file.buffer.toString());

        const results = await Promise.allSettled(
          shipments.map(s => this.createShipment(s, req.
          partner))
        );

        res.status(207).json({
          total: results.length,
          successful: results.filter(r => r.status ===
          'fulfilled').length,
          failed: results.filter(r => r.status === 'rejected').
          length
        });
      }
    );
  }
}
```



Chapter 4

The ROI of code modernization

Chapter 4

The ROI of code modernization

While the costs of modernization are often front-loaded and visible, the returns manifest across multiple dimensions that compound over time, including increased speed and scale of development, risk reduction, improved knowledge preservation and financial gains.

Speed and scale

Development teams experience dramatic acceleration in feature delivery when freed from the constraints of legacy systems. Multi-month projects compress to weeks through the elimination of technical barriers that previously required specialized knowledge and careful orchestration. By leveraging modern programming languages, frameworks and architectures, organizations create applications that handle increased workloads more efficiently than their legacy counterparts, often with fewer resources and lower operational costs.

Risk reduction

Modern systems incorporate current security practices and receive regular updates from active vendor and open source communities. The incorporation of latest security practices and technologies better protects sensitive data and customer information from increasingly sophisticated cyber threats. High-availability architectures eliminate single points of failure that plague legacy systems, where a single component failure could bring down entire business operations for hours or days.

Knowledge preservation

Claude Code and other agentic coding tools can generate documentation with the run of a single command or text prompt, providing new developers with the context they need to understand and maintain systems effectively. Using Claude Code, teams can extract institutional knowledge from your legacy codebase, create understandable documentation and then facilitate migration and Q&A through robust test suites. This process prevents the cycle from repeating, ensuring that codebases and the critical systems powered by them can easily migrate to more modern architectures.

Financial impact

Most significantly, code modernization can translate to financial gains. A recent **Deloitte study** suggests that companies who embrace digital transformation initiatives like code modernization have a 14% higher market value than those that don't, while the Harvard Business Review argues that modern architectures **increase company valuations** by reducing technical risks. By reducing operational expenses through decreased maintenance costs and reduced need for specialized expertise, teams that embrace code modernization can spend more resources innovating.

Chapter 5

Choosing the right agentic coding solution

Chapter 5

Choosing the right agentic coding solution

When evaluating agentic coding solutions for code modernization, organizations must assess critical capabilities that determine whether a tool can handle the complexity of real-world legacy systems.

Deep codebase awareness

Effective agentic coding tools demonstrate the ability to understand and navigate complex codebases without requiring extensive manual setup. These tools should run multi-file tasks using deep codebase awareness that automatically understands project structures, dependencies and architectural patterns, enabling consistent changes across entire codebases. Advanced tools like Claude Code can trace execution paths through multiple files and frameworks, understanding how changes propagate through systems and identifying potential impacts before they become problems.

Working with all your tools

Agentic coding solutions must integrate seamlessly with existing development workflows and toolchains. Version control systems, testing frameworks, build tools and deployment pipelines should work naturally with AI assistance, requiring no changes to established processes. Integration extends beyond development tools to encompass the entire software delivery lifecycle, connecting with project management systems, testing platforms and monitoring systems to ensure AI assistance enhances rather than disrupts existing workflows.

Running anywhere: flexible deployment

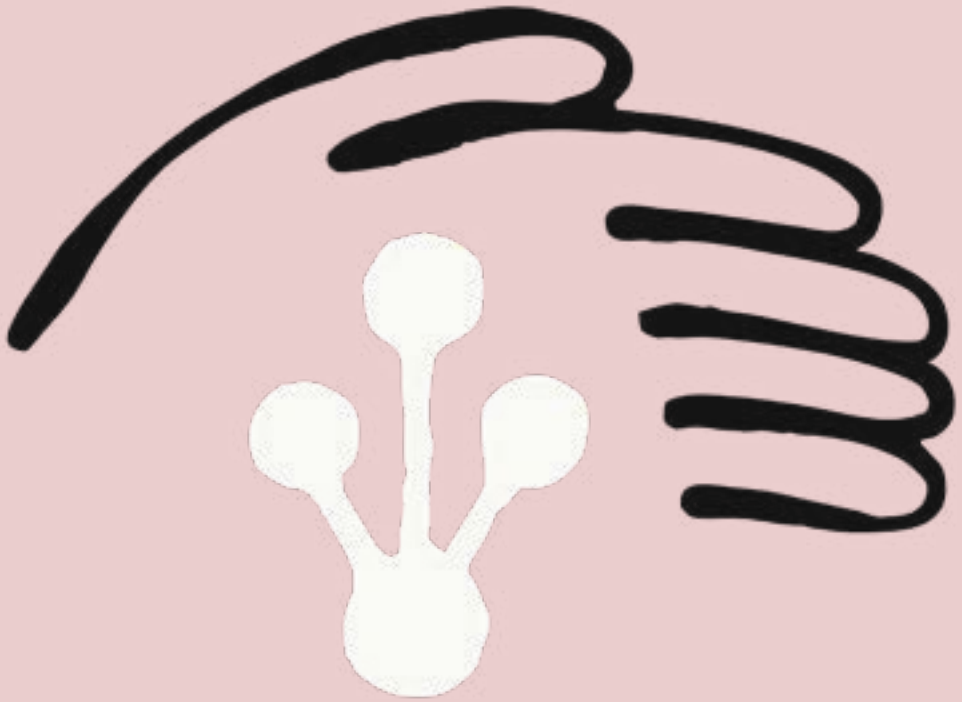
Enterprise development teams require flexibility in deployment options. Tools must operate effectively in terminals, integrate with IDEs or run as headless CLI tools for automation scenarios. Solutions like Claude Code provide integration with popular development environments including Visual Studio Code, JetBrains IDEs and Neovim. Cloud-based and local deployment options ensure that all organizations can benefit from AI assistance regardless of their security posture or infrastructure constraints.

Secure by design

Enterprise-grade security ensures that code and queries go directly to AI models via encrypted APIs without passing through intermediary servers. Tools demonstrate their work transparently, showing planned changes and requesting approval before modifying code. Comprehensive audit trails and role-based access controls maintain governance and compliance requirements. Integration with secure cloud enterprise AI deployments, including Amazon Bedrock and Google Vertex AI, provides flexibility while maintaining security standards.

Advanced features

Effective agentic coding solutions comprise three essential technical components: memory systems for maintaining context across long-running projects, function capabilities for executing complex transformations and the ability to connect to many tools through open standards like the **Model Context Protocol (MCP)**. These components work synergistically to ensure successful modernization outcomes.



Chapter 6

Getting started with code modernization

Chapter 6

Getting started with code modernization

Successful code modernization requires more than technical expertise—it demands strategic planning and honest assessment of organizational capabilities. Before embarking on transformation initiatives, organizations must understand their current state, available resources and potential roadblocks to ensure modernization efforts deliver meaningful business value rather than creating additional technical debt.

Assessing organizational readiness

Organizations considering code modernization should evaluate their current situation across several key dimensions to determine if the investment will deliver sufficient value. As a general rule of thumb, teams spending more than 40% of their time on maintenance activities rather than new feature development face a clear indicator that modernization could dramatically improve productivity. The challenge of hiring and onboarding engineers for legacy technologies provides another signal—when recruiting takes months and new hires require extensive training on obsolete languages, the cost of maintaining status quo often exceeds modernization investment.

Choosing the right implementation approach

Organizations beginning their modernization journey should first assess their testing and validation capabilities. Those with comprehensive unit tests or evaluation suites can validate that modernized code maintains functional equivalence with legacy systems. Teams without existing tests should identify critical modules and work with domain experts (we recommend a professional services partner) to build test suites using AI assistance before attempting migration. Claude Code can accelerate test creation by analyzing existing code to understand expected behaviors and edge cases.

Development teams might also begin by refactoring non-critical modules or migrating systems with fewer dependencies to build confidence and experience. This iterative approach allows organizations to learn and refine their modernization practices before tackling core business systems. Success in early projects builds organizational support and provides concrete examples of value delivery that justify larger investments.

Building momentum through phases

The journey from initial pilot to enterprise-wide transformation typically follows a predictable pattern that organizations can plan around. This phased approach allows teams to build confidence and expertise while minimizing risk. See below for an example:

Timeline	Phase and activities
Weeks 1-2	Focus on identifying a low-risk, high-visibility legacy system that can demonstrate modernization value while building organizational confidence. Target systems like batch reporting modules, standalone calculation engines or peripheral integration services—complex enough to showcase AI capabilities but isolated enough that issues won't impact core operations. During this phase, conduct code archaeology: analyze COBOL/mainframe job flows, map data dependencies and document business rules embedded in old subroutines.
Weeks 3-4	Deploy Claude Code to analyze legacy code and demonstrate modernization potential. AI agents read through COBOL programs, JCL scripts and VSAM file definitions, extracting business logic and documenting hidden dependencies. Generate initial code translations to Java or Python, create data flow diagrams and identify technical debt patterns. Build a proof of concept that modernizes a critical subroutine or batch job, showing side-by-side execution with identical outputs. This phase reveals complexities like implicit date handling, packed decimal conversions and undocumented business rules—providing crucial insights for production migration.
Weeks 5-8	Complete the first full system migration from mainframe to cloud. AI agents handle code translation while preserving exact business logic, generate comprehensive test suites covering edge cases found in production data, create API wrappers for gradual transition and produce documentation explaining every transformation decision. Implement parallel run capabilities where modernized code operates alongside legacy systems, comparing outputs for validation. Address challenges like EBCDIC to ASCII conversions, hierarchical to relational data transformations and CICS transaction reimplementation. By week 8, achieve production cutover with the legacy system decommissioned.
Month 3+	Expand modernization efforts based on proven patterns. Teams tackle increasingly complex systems—i.e. core banking engines, real-time trading platforms, integrated policy management systems. AI agents now work with accumulated knowledge: reusing proven conversion patterns, identifying common anti-patterns before they cause issues and suggesting architectural improvements beyond mere translation. Establish a modernization factory approach: parallel teams working on different systems, shared libraries of conversion utilities, automated testing frameworks and continuous knowledge capture.

Treating AI like a thought partner

Modern AI tools excel when engaged as thought partners in architectural discussions that go beyond simple code generation. Agentic coding tools like Claude Code demonstrate particular strength in teaching concepts, debugging complex issues, and supporting long-form thinking about system design. Rather than simply generating code, these tools walk developers through the reasoning behind architectural decisions, helping teams understand not just what changes to make but why those changes improve the system. They enable newer developers to architect systems and perform modernizations without have an expert-level understanding of the codebase, allowing all team members to meaningfully contribute.

As teams work with agentic coding tools like Claude Code, they also benefit from automatic edge case discovery and test generation that improve quality while reducing manual burden. The ability to quickly explore design alternatives helps teams make better architectural decisions by evaluating multiple approaches and understanding their trade-offs before committing to implementation.

The modernization maturity model

Organizations typically progress through four distinct levels of modernization maturity, each building capabilities for more sophisticated approaches. Understanding these maturity levels helps organizations assess their current state and chart a path toward more effective modernization practices. Each level builds on the previous, creating a pathway for organizations to evolve their modernization capabilities over time. See below for an example:

Maturity level	Characteristics
Ad hoc	Legacy systems addressed only during failures—retiring COBOL programmers, dying mainframes or regulatory deadlines. Teams scramble to decode undocumented code and patch 30-year-old systems. No documentation, test suites or transition plans exist. Knowledge lives solely in retiring experts' heads. Technical debt compounds as teams add workarounds to avoid touching core legacy code.
Planned	Annual projects target specific systems, though selection reflects politics over business value. Teams plan COBOL-to-Java conversions with budgets and timelines, but efforts remain siloed—each project reinvents the wheel. Basic legacy inventories exist (lines of code, age, criticality) without sophisticated ROI analysis. Some automated conversion tools help, but humans still manually verify most translations.
Systematic	Dedicated teams follow standardized processes with clear metrics. Living inventories track technical debt scores and modernization readiness. AI tools continuously analyze legacy code, documenting business rules and suggesting refactoring. Established playbooks guide common patterns: i.e. batch-to-streaming, monolith decomposition. Automated testing ensures compatibility. Every COBOL subroutine documented, every dependency mapped.
Optimized	AI agents proactively scan systems and generate modernization proposals with ROI analysis. Claude reads COBOL, understands intent and automatically creates cloud-native microservices with full test coverage. Continuous background modernization: i.e. removing dead code, optimizing queries, refactoring to modern frameworks. When regulations change, AI automatically generates compliance updates. Humans focus on strategy while AI handles routine transformations.

Resources & next steps

Technical leaders ready to begin their modernization journey can access comprehensive resources and support through the following resources:

Detailed information about Claude Code and its capabilities for code modernization.

Technical documentation covering implementation patterns, best practices and integration guides.

How Anthropic teams use Claude Code provides insights into how Anthropic employees across functions use Claude Code to refactor code, debug systems and other critical engineering tasks.

With agentic coding tools at your fingertips, transforming legacy codebases from technical debt into your strategic advantage has never been more achievable.

Reach out to Anthropic's Sales team to learn more.

Claude