Scaling Agentic Coding Across Your Organization

How to transform your engineering organization with Claude Code, from pilot to production.

Claude

Contents

Foreword

How to transform your engineering organization with Claude Code, from pilot to production.

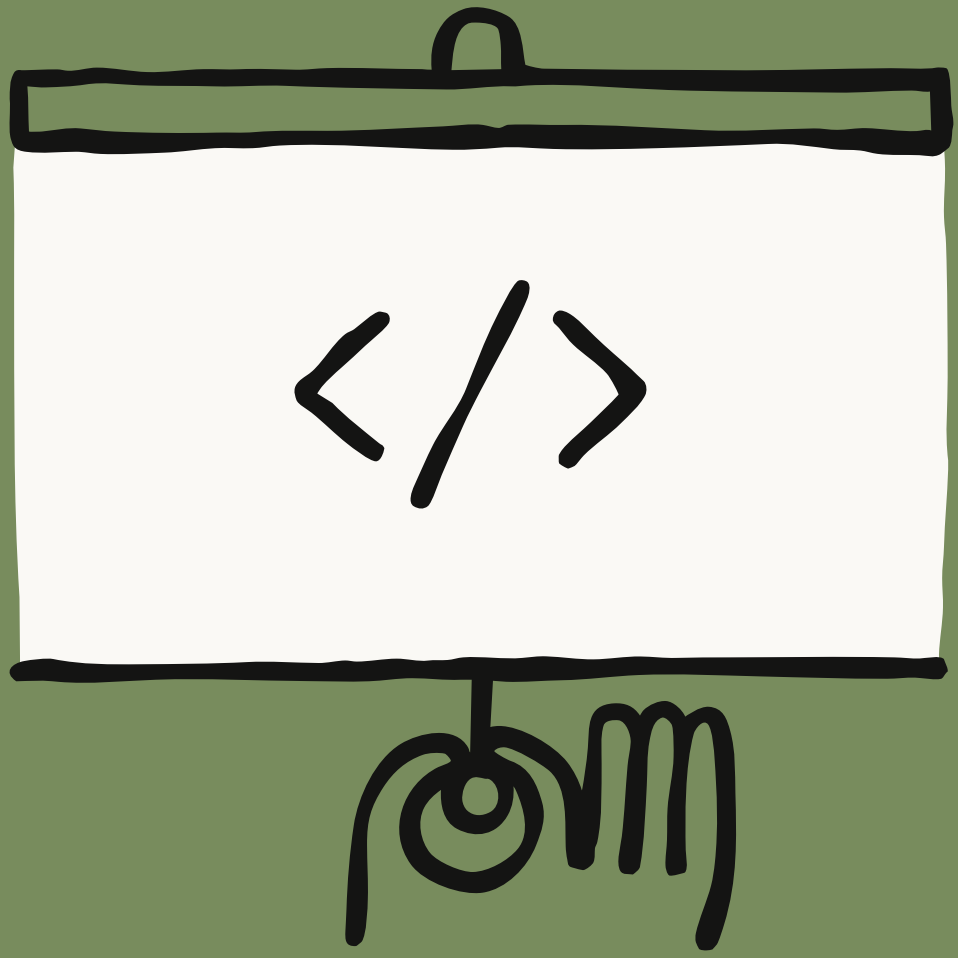
Over the last year, agentic coding tools like Claude Code have become table stakes for engineers, serving as a force multiplier for organizations of all sizes. In this new landscape, technical leaders face a critical question: How do you scale agentic coding tools from a handful of early adopters to an entire engineering organization?

The stakes have never been higher. Companies that successfully deploy agentic coding at scale are transforming how their teams work—enabling faster development cycles and freeing engineers to focus on architecture and innovation rather than repetitive implementation tasks. Meanwhile, organizations that approach this transformation haphazardly often face adoption resistance, inconsistent results, and missed opportunities to fundamentally reimagine how software gets built.

The good news is that successful patterns have emerged. Through working with engineering teams across various industries and company sizes, we've identified the critical factors that separate thriving agentic engineering cultures from those that struggle. These insights reveal that success isn't just about choosing the right tool; it's about thoughtfully orchestrating changes to workflows, team dynamics, skill development, and even how you measure engineering success.

Based on insights from Anthropic's Applied AI team and lessons learned from our customers, this guide shares best practices for rolling out agentic coding tools across your organization.

Let's dive in.



Chapter 1

What is agentic coding?

Chapter 1

What is agentic coding?

Agentic coding tools represent a leap beyond traditional AI code generation. These are AI systems that act as autonomous coding agents, capable of understanding context, planning approaches, and executing entire coding tasks with minimal human oversight. Agentic coding tools can:

Modernize legacy codebases

Teams across industries can more easily migrate COBOL, SAS, and other outdated systems to modern backends, turning multi-year projects into months-long initiatives.

Accelerate onboarding

New engineers can explore and understand complex codebases by simply asking questions about architecture, dependencies, and implementation details.

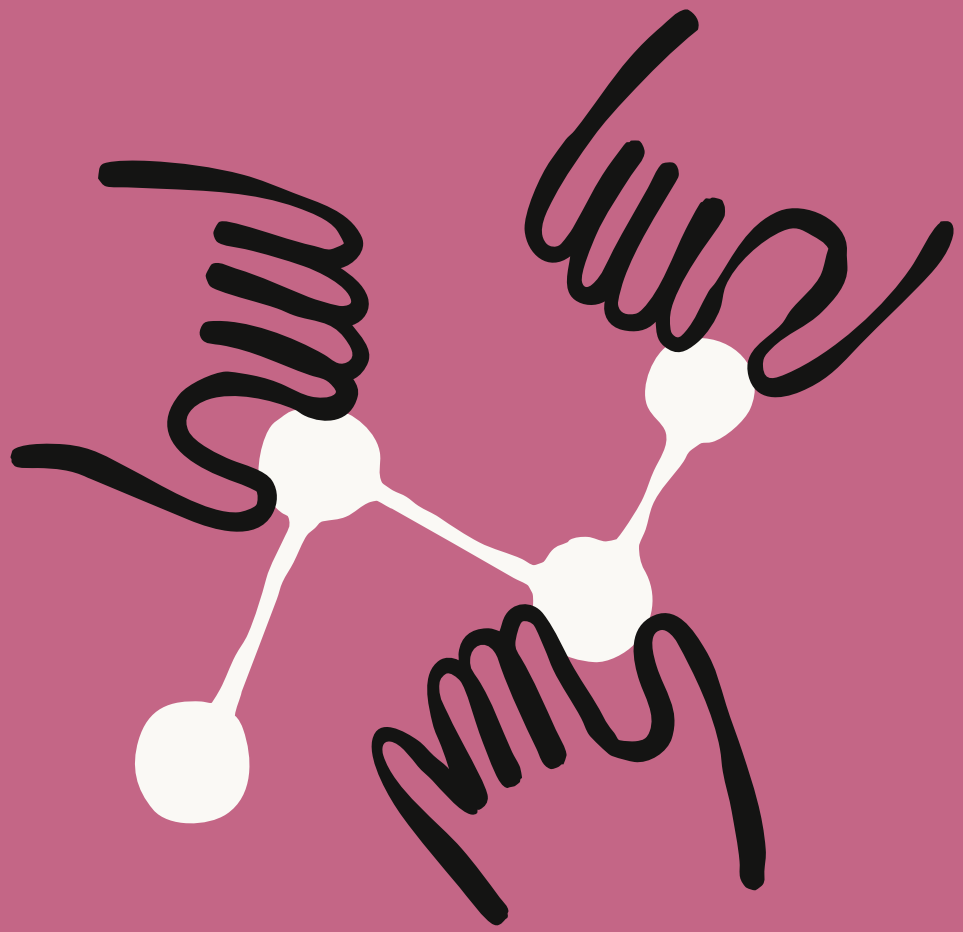
Automate incident response

Site Reliability and DevOps teams can build custom agents that automatically triage and resolve on-call incidents.

Enable non-technical contributors

Product managers can write requirements informed by actual codebase constraints, while designers can turn Figma mockups into functional prototypes.

And these use cases only scratch the surface of what's possible. Check out our article on how Anthropic teams use Claude Code for additional use cases and examples.



Chapter 2

Rolling out agentic coding at your organization

Chapter 2

Rolling out agentic coding at your organization

The most successful agentic coding rollouts follow a deliberate expansion strategy that builds expertise and enthusiasm organically. Here's what we've seen work:

Start with power users

Begin with a pilot group of 20-50 developers who are already comfortable with AI-assisted tools. These aren't just your early adopters—they're your future agentic coding champions. Task them with:

- Ask them to spend time using Claude Code for common use cases. This is the best way to realize what customizations are useful and make sure the tool works well with your codebase.
- Creating custom slash commands in Claude Code tailored to your organization's codebase and standards, such as /migrate-db for database schema updates, /add-feature with company-specific boilerplate, and /fix-security for common vulnerability patches.
- Creating documentation files with your coding standards, outlining gotchas, and highlighting best practices via CLAUDE.md files, special files that Claude automatically pulls into context when starting a new conversation (Remember: take time to iterate on your CLAUDE.md files as you would any other agentic prompt. The devil is in the details!)
- Identifying repetitive coding workflows that Claude Code can automate by engaging with development teams, DevOps engineers, and technical leads across the organization to understand their pain points - such as boilerplate code generation, test suite creation, documentation updates, code refactoring tasks, API client generation, dependency updates, and routine bug fixes that consume significant developer time
- Establish a dedicated Slack or Teams channel for sharing best practices, troubleshooting issues, and broader discussion
- Build wrapper scripts to handle authentication for third-party tools, like AWS and GCP

For more tips, check out our **Eng Blog article** on Claude Code best practices.

Launch with a hackathon

Rather than a phased rollout that leaves teams waiting their turn, unite your organization with a kick-off hackathon. Your pilot users become mentors, sharing prompts and techniques while everyone learns together. This creates network effects that accelerate adoption and organic learnings. Bonus points if you order pizzas to stave off hunger and drive participation.

Scale through internal expertise

As adoption grows, your pilot users evolve into internal consultants. Eventually, your organization will get to the place where they can run their own agentic coding workshops, with early adopters leading sessions and creating ongoing educational content inspired by their own learnings.

Sharing CLAUDE.md files

CLAUDE.md files are an ideal place for documenting repository etiquette, developer environment setup, and any unexpected behaviors particular

to a given project. However, the real power of CLAUDE.md files emerges when they're shared strategically across teams and organizations, creating a knowledge layer that scales AI-powered development. Here are some best practices for using them at scale:

Create project-level CLAUDE.md files: Name it CLAUDE.md and check it into git so that you can share it across sessions and with your team. This simple practice transforms individual Claude Code sessions into team-aligned development experiences.

Commit to main branch: Place CLAUDE.md in your repository root and commit it to your main branch. This ensures every developer who clones the repository automatically inherits the project's Claude Code configuration and context.

Include in onboarding checklists: Make reviewing and understanding the project's CLAUDE.md file part of your developer onboarding process. New team members should understand not just the codebase, but how Claude Code should be used within the project context.

Version control like documentation: Treat CLAUDE.md changes with the same rigor as documentation updates. Include updates in pull requests when architectural decisions change, new development patterns emerge, or team conventions evolve.

Branch-specific variations: For projects with significantly different development patterns across branches (feature branches, release branches), consider branch-specific CLAUDE.md content that reflects the current development focus.

See the **appendix** for an example project-level **CLAUDE.md** structure.

Chapter 3

Measuring ROI

Chapter 3

Measuring ROI

Pilots are only impactful if they drive value and highlight the art of the possible. For most engineering leaders, “how do we measure ROI?” remains the top question to answer when it comes to driving wider adoption.

Beyond simply measuring lines of code (which is a notoriously a spotty metric), teams are finding both quantitative and qualitative ways to quantify agentic coding’s impact:

Sprint throughput: Teams with mature DevOps practices correlate agentic coding adoption with feature delivery speed

Time-to-completion: Track how long standard tasks take before and after adoption

Migration acceleration: Reduced time to migrate off legacy codebases onto modern systems

Developer satisfaction surveys: Reduced time spent on repetitive tasks and increased time spent on more thoughtful design work

Onboarding speed: Increased time to productivity for new hires

Cross-team collaboration: Less reliance on developers to prototype and test products before deployment

Additionally, Claude Code comes baked with its own productivity metrics tracking via Activity Metrics. Insights include:

Lines of code accepted: Track the actual code volume that developers are accepting and using from Claude Code

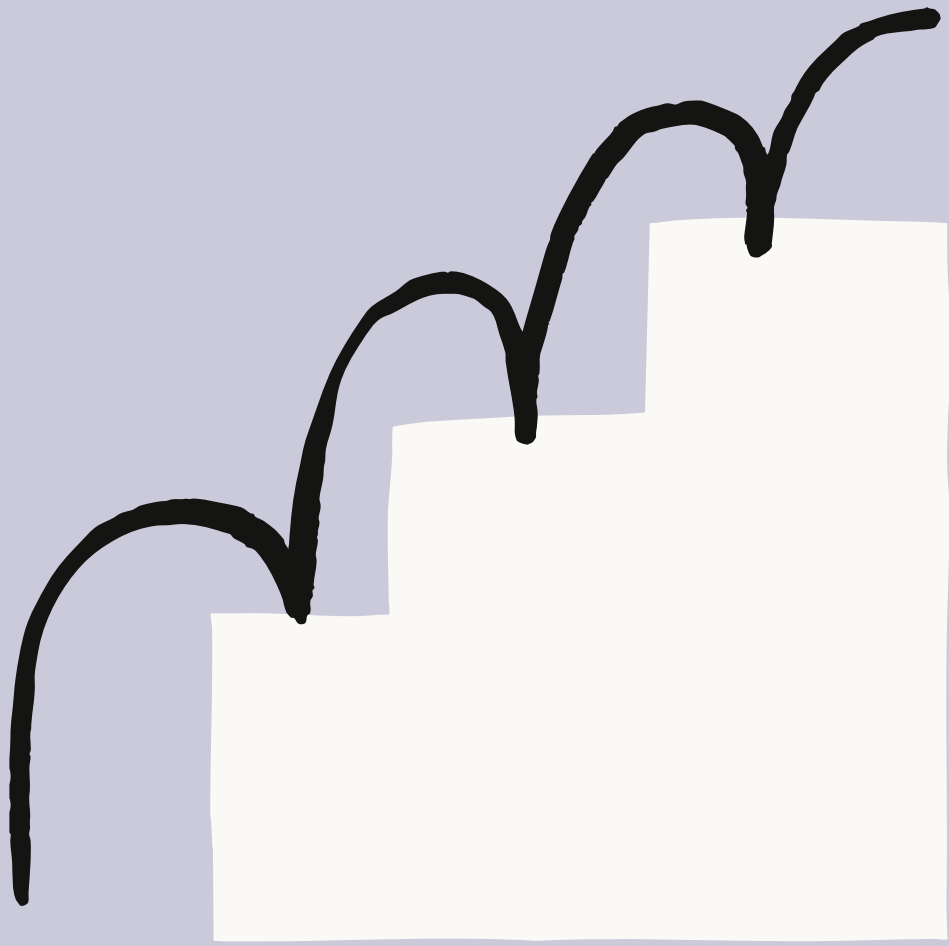
Suggestion accept rate: Understand how well Claude Code’s suggestions align with your team’s needs

Activity trends: Monitor daily active users and sessions to see adoption patterns

Spend tracking: View organization-wide and per-user spending with daily granularity

Team insights: See individual developer metrics including avg daily spend and lines per day

At the end of the day, the most compelling metric might be the simplest, i.e.: “This task used to take our team a week. Now one engineer completes it in half a day.”



Chapter 4

Avoiding common challenges to adoption

Chapter 4

Avoiding common challenges to adoption

Like any technology, agentic coding tools can't be deployed successfully in a vacuum. Here's how to sidestep the most common adoption challenges and ensure your Claude Code roll out drives impact, fast:

Don't fall into the "do everything" trap

Agentic tools are enthusiastic, but don't always have enough context to be impactful. New users often give them massive, unbounded tasks with poor results. The solution? Test-driven development (TTD), which provides guardrails and clear success criteria for your agentic coding tool.

Start with test specifications: Instead of asking Claude Code to "build a user authentication system," begin by having it write comprehensive tests first. Ask it to create test cases that define exactly what success looks like: authentication flows, edge cases, error handling, and security requirements.

Implement features incrementally: Break down your request into small, testable chunks. Have Claude Code implement just enough code to make one test pass at a time. For example, start with basic login validation, then add password hashing, then session management.

Validate at each checkpoint: After each implementation step, run the tests and review the code changes. Claude Code can help you analyze test results and identify issues, but don't let it move forward until the current functionality is solid.

Expand scope gradually: Once core functionality is working and tested, incrementally add new requirements. Ask Claude Code to first write tests for the new feature, then implement it. This prevents scope creep and maintains code quality.

Use Claude Code's iterative nature: Take advantage of the command-line workflow by running focused commands like "write tests for user registration" followed by "implement the registration logic to pass these tests" rather than one massive "build everything" request.

Check out the **appendix** for an example of how to run test-driven development with Claude Code.

Overcome the context gap

"This isn't working" or "The button is too big" gives your AI nothing to work with. These vague descriptions lead to wasted iterations and frustrated debugging sessions. Instead, we suggest veering on the side of over-sharing context with Claude and providing clear, actionable feedback for optimal results.

Share comprehensive error information: Instead of "it crashed," provide the full error message, stack trace, and the specific action that triggered it. Copy-paste terminal output, browser console errors, or IDE error panels directly into your Claude Code session.

Document the complete environment: Include your operating system, language versions, framework details, and relevant dependencies. Claude Code needs to understand your technical stack to provide accurate solutions.

Use visual debugging strategically: When dealing with UI issues, take screenshots and describe exactly what's wrong—"the login button extends 20px beyond the container border on mobile screens" vs. "the button looks weird." For command-line tools, share before/after terminal outputs.

Specify precise expected vs. actual behavior: Write clear acceptance criteria like "Expected: API returns 200 status with user data. Actual: Returns 401 with 'invalid token' message." This gives Claude Code concrete targets to work toward.

Include relevant file contents: Share the specific code files, configuration files, or data that relate to your issue. Claude Code can't debug code it can't see.

Check out the **appendix** for a sample prompt that offers ample context for Claude to (hopefully) zero in on the root cause of a software bug.

Prioritize prompt engineering

Success with agentic coding requires learning to communicate effectively with AI. Many developers jump in expecting Claude Code to read their minds, then get frustrated with subpar results. Additionally, as with any AI agent, providing the right structure, contents, and order is critical for ensuring optimal results.

Mastering Claude Code communication

Treat Claude like an engineer: Ask yourself if your teammate would understand exactly what you're asking for, based on the prompt you're giving them. If not, try to anticipate what questions they might have next or what things they'd want clarification or more detail on before they start working, and provide them to claude proactively.

Use technical precision: Replace vague terms with specific technical language. Instead of "make it faster," specify "optimize the database query to reduce response time from 2 seconds to under 500ms" or "implement caching to reduce API calls."

Provide examples and constraints: Show Claude Code what success looks like with concrete examples. "Follow this existing API pattern [paste code]" or "Use this coding style [share style guide]" gives clearer direction than abstract requirements.

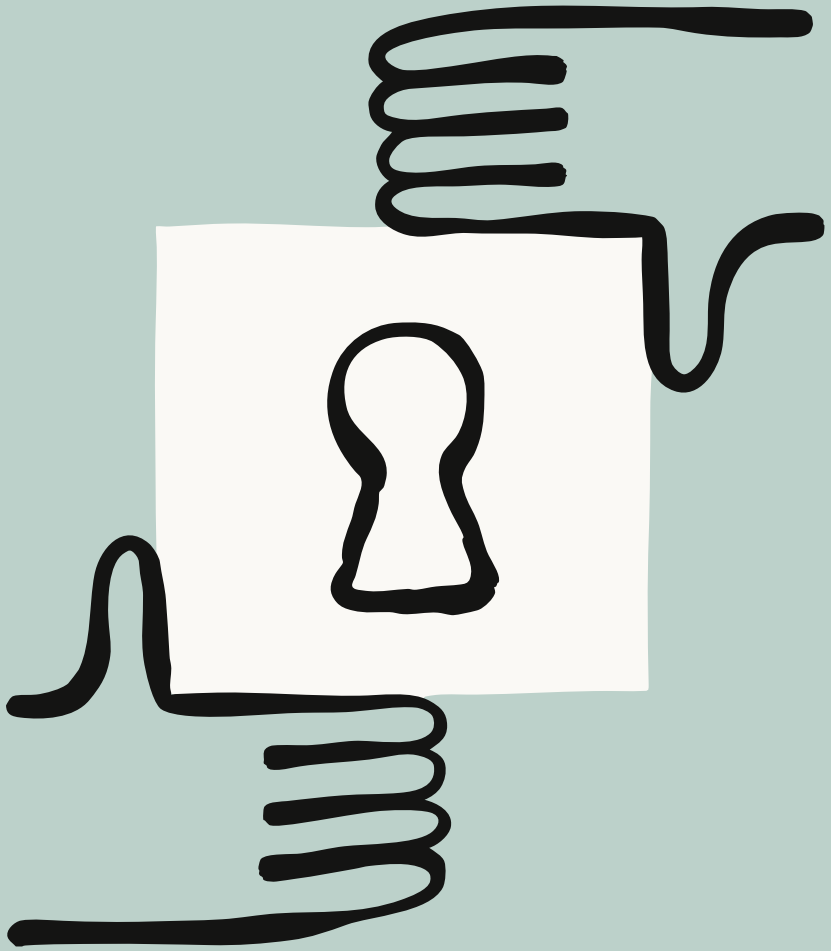
Break complex tasks into focused commands: Rather than "build a complete e-commerce system," use sequential prompts: "Create the database schema," then "implement product catalog API," then "add shopping cart functionality." Each command should have a single, clear objective.

Learn incremental refinement: Start with basic functionality and iteratively improve. "Create a simple user login form" followed by "add input validation" then "implement password strength requirements" builds better results than trying to specify everything upfront.

Master the feedback loop: Learn to give Claude Code specific feedback on its output. "The error handling is too generic—add specific validation for email format and password length" guides better improvements than "fix the validation."

Practice context management: Understand what information Claude Code retains within a session and what needs to be re-stated. Reference previous work explicitly: "Using the authentication middleware from earlier, now add role-based permissions."

Check out the **appendix** for a sample prompt with an effective structure for ensuring adequate output.



Chapter 5

Security first: protecting your codebase

Chapter 5

Security first: protecting your codebase

Security considerations aren't optional. They're foundational to successful agentic coding adoption. Unlike traditional development where security reviews happen at the end, agentic tools require security-by-design thinking from day one.

While Claude Code can help write more secure code, we recommend using it alongside your team's existing security tools, rather than replacing them.

Policy-as-code enforcement: Create explicit security policies in your CLAUDE.md files that Claude Code can reference. For example: "All database queries must use parameterized statements," "API endpoints require authentication middleware," or "No hardcoded secrets in configuration files."

Security review automation: Use a /security-review command to run ad-hoc security analyses from your terminal before committing code. This command uses a specialized security-focused prompt that checks for common vulnerability patterns including SQL injection risks, cross-site scripting (XSS) vulnerabilities, authentication and authorization flaws, insecure data handling, and dependency vulnerabilities. You can also configure a /security-review slash command that audits GitHub Actions workflows for security vulnerabilities by checking against policies defined in your CLAUDE.md files.

Security pattern libraries: Build a repository of approved security patterns that Claude Code can reference. Instead of letting it improvise authentication logic, provide tested, compliant examples it should follow.

Automated vulnerability remediation: Use Claude Code to not just identify security issues but propose specific fixes. "This SQL query is vulnerable to injection—here's the parameterized version" becomes part of its standard response pattern.

Configuring enterprise permissions: With Claude Code, admins can push out configurations and permitted MCP tools to all users through enterprise managed policy settings that take precedence over user and project settings. This centralized approach transforms Claude Code from a powerful but potentially inconsistent individual tool into a governed, enterprise-ready development platform.

MCP server management

Model context protocol is an open standard released by Anthropic that standardizes how AI models connect with external data sources and tools. By integrating MCP servers with Claude Code, you can add additional context and functionality to your software development environment. Still, while the agentic coding ecosystem moves fast, not all MCP server integrations meet enterprise security standards. Here's how to keep your MCP servers up-to-date:

Security-first evaluation process: Before approving any MCP server, conduct thorough security assessments. Evaluate data handling, API security, access controls, and vendor security posture. Create a standardized evaluation rubric that covers code access, data transmission, and third-party dependencies.

Curated integration marketplace: Create an internal "app store" of pre-approved MCP servers. Include security assessments, approved use cases, and implementation guidelines for each tool. This prevents shadow IT adoption while enabling innovation.

Integration sandboxing: Test new MCP servers in isolated environments before production deployment. Monitor data flow, API calls, and security behavior to identify potential risks before they impact your main codebase.

Regular security audits: Schedule quarterly reviews of all approved MCP servers. Check for security updates, vulnerability reports, and changes in vendor security practices. Maintain an integration lifecycle that includes deprecation plans for tools that don't meet evolving security standards

Code review evolution

Organizations can use agentic coding tools themselves as security reviewers. This approach can identify potential vulnerabilities and suggest secure alternatives as it relates to the following use cases (and more):

Multi-layered security analysis: Configure Claude Code to perform different types of security reviews—static analysis for common vulnerabilities, architectural review for security patterns, and compliance checking against your internal policies.

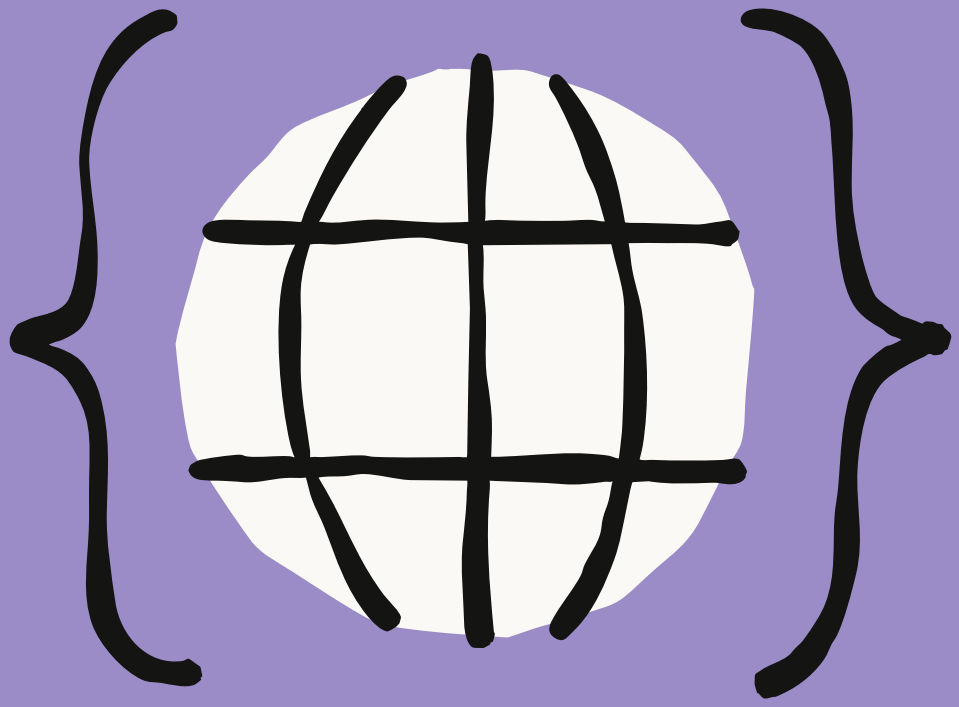
Context-aware vulnerability detection: Train Claude Code to understand your specific technology stack and security requirements. A Node.js application has different security considerations than a Python Flask app or a Go microservice.

Secure alternative suggestions: When Claude Code identifies a security issue, require it to propose specific, tested alternatives. "This cookie handling is insecure" should come with "Use httpOnly and secure flags with SameSite=Strict configuration."

Security debt tracking: Use Claude Code to identify and catalog technical security debt—outdated dependencies, deprecated security practices, or missing security controls—and prioritize remediation efforts.

Collaborative security education: Let Claude Code explain security issues in developer-friendly terms, turning each security review into a learning opportunity rather than just a compliance checkpoint.

With these best practices in place, Claude Code can help improve your security posture and accelerate code auditing and reviews.



Chapter 6

Beyond the obvious: innovative agentic coding applications

Chapter 6

Beyond the obvious: innovative agentic coding applications

The most exciting agentic coding use cases often come from unexpected places. Organizations like Brex and Rakuten are discovering that Claude Code's capabilities extend far beyond basic code generation into transformative business processes.

Teams are building specialized agents on top of agentic coding platforms like the Claude Code SDK, creating domain-specific AI assistants that understand their unique technical challenges and business requirements. Here are a few examples:

Intelligent on-call automation: Deploy agents that monitor system health, parse complex log files, correlate errors across microservices, and implement targeted fixes. These agents can handle routine incidents like memory leaks, database connection issues, or service scaling, escalating only when human judgment is required.

Migration and modernization specialists: Create purpose-built agents for specific technology transitions. A legacy-to-cloud agent might understand your existing mainframe COBOL patterns and systematically convert them to containerized microservices, maintaining business logic while updating architecture.

Continuous documentation engines: Develop agents that monitor code changes, understand architectural decisions, and automatically update technical documentation. These agents can maintain API documentation, system diagrams, and runbooks in sync with actual implementation.

Performance optimization agents: Build specialized agents that analyze application performance, identify bottlenecks, and implement optimizations. They can monitor database query performance, suggest indexing strategies, or optimize algorithm implementations based on production metrics.

Regulatory compliance agents: Create agents that understand specific regulatory requirements (GDPR, SOX, HIPAA) and continuously audit codebases for compliance issues, automatically implementing required controls and documentation.

What types of agents would you build?

Chapter 7

Looking forward

Chapter 7

Looking forward

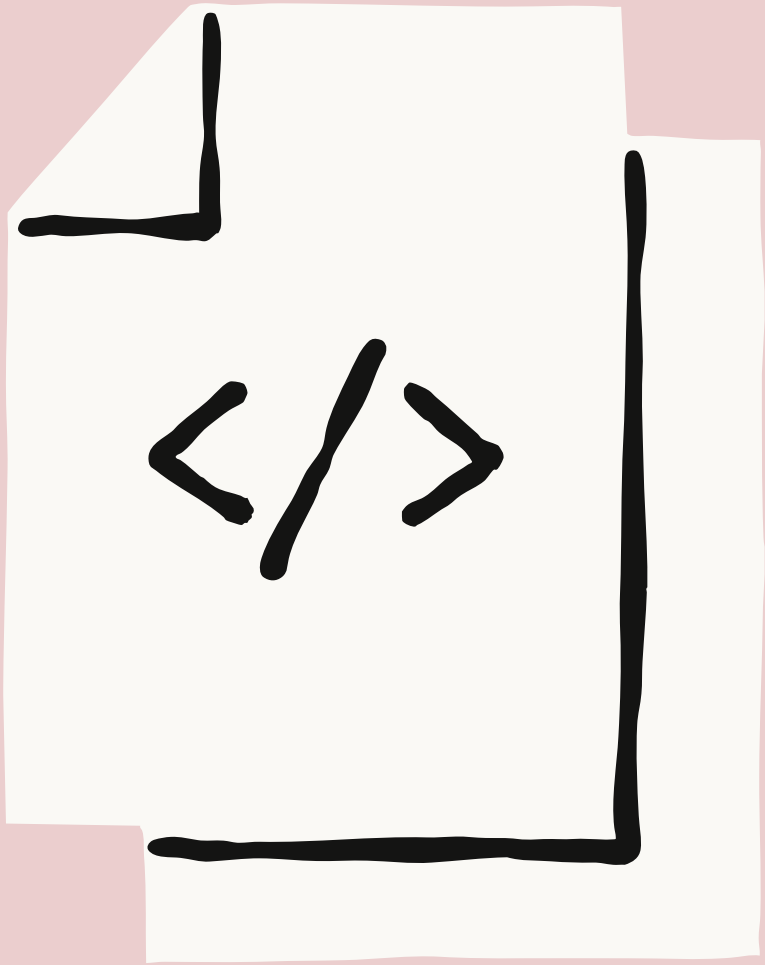
Agentic coding represents a fundamental shift in how we build software, from human-written to human-guided development. The organizations seeing the most success aren't those who move fastest, but those who build the strongest foundations.

Start small with a pilot group. Build your internal expertise. Create the infrastructure for success. Then scale deliberately, using hackathons and internal champions to drive adoption.

Agentic coding is your team's cognitive multiplier. The only question is: how quickly can you help them harness it?

Ready to scale agentic coding across your organization? Reach out *to our team to learn more.*

Note: As agentic coding evolves rapidly, best practices continue to emerge. This guide reflects current insights as of August 2025.



Appendix

Appendix

Project-level CLAUDE.md structure example

Example project-level CLAUDE.md structure

```
# Project Feline - Cloud Task Scheduler

## Development Environment
- Use Node.js 18+ with nvm for version management
- Run `npm run setup` for initial environment configuration
- All database changes require migration files in `/migrations`

## Claude Code Guidelines
- Always run tests before committing: `npm test`
- Use TypeScript for all new modules
- Follow our API naming convention: `/api/v2/tasks/{resource}`
- Security: Never commit API keys or database credentials in
code

## Architecture Notes
- Task scheduling uses Redis queues for job management
- All external API calls must include retry logic and timeouts
- Database queries should use connection pooling for
performance

## Current Sprint Focus
- Implementing distributed task execution
- Focus on fault-tolerance and recovery patterns
- All scheduling logic requires unit tests + integration tests
```

Test-driven development with Claude Code: User authentication system example

Step 1: Define requirements with Claude Code

Bash

```
# Terminal command to Claude Code

claude-code --task "Create comprehensive tests for a user
authentication system with registration, login, password
reset, and JWT token management"
```

Claude code response

Python

```
# tests/test_auth_system.py
import pytest
import jwt
from datetime import datetime, timedelta
from unittest.mock import Mock, patch
from auth_system import AuthSystem, User, AuthenticationError,
ValidationError
class TestUserAuthentication:
    @pytest.fixture
    def auth_system(self):
        return AuthSystem(secret_key="test_secret", db_
connection=Mock())

    @pytest.fixture
    def valid_user_data(self):
        return {
            "email": "user@example.com",
            "password": "SecurePass123!",
            "confirm_password": "SecurePass123!"
        }

    # Registration Tests
    def test_register_user_success(self, auth_system, valid_
user_data):
        """Test successful user registration"""
        with patch.object(auth_system.db, 'user_exists',
return_value=False):
            with patch.object(auth_system.db, 'create_user',
return_value=True):
                result = auth_system.register(**valid_user_data)
                assert result['success'] == True
                assert result['user_id'] is not None
                assert result['message'] == "User registered
successfully"

    def test_register_user_duplicate_email(self, auth_system,
valid_user_data):
        """Test registration with existing email"""
        with patch.object(auth_system.db, 'user_exists',
return_value=True):
    def test_register_invalid_email_format(self, auth_system):
          """Test registration with invalid email format"""
          invalid_data = {
              "email": "invalid-email",
              "password": "SecurePass123!",
              "confirm_password": "SecurePass123!"
```

```
        }
        with pytest.raises(ValidationError) as exc_info:
            auth_system.register(**invalid_data)
        assert "Invalid email format" in str(exc_info.value)

    def test_register_weak_password(self, auth_system):
        """Test registration with weak password"""
        weak_password_data = {
            "email": "user@example.com",
            "password": "123",
            "confirm_password": "123"
        }
        with pytest.raises(ValidationError) as exc_info:
            auth_system.register(**weak_password_data)
        assert "Password must be at least 8 characters" in
 str(exc_info.value)

    # Login Tests
    def test_login_success(self, auth_system):
        """Test successful login with valid credentials"""
        mock_user = Mock()
        mock_user.id = 1
        mock_user.email = "user@example.com"
        mock_user.password_hash = "$2b$12$hashed_password"

        with patch.object(auth_system.db, 'get_user_by_email',
return_value=mock_user):
            with patch.object(auth_system, '_verify_password',
return_value=True):
                result = auth_system.login("user@example.com",
"correct_password")
                assert result['success'] == True
                assert 'access_token' in result
                assert 'refresh_token' in result
                assert result['user']['email'] == "user@
example.com"
```

```
    def test_login_invalid_credentials(self, auth_system):
        """Test login with invalid credentials"""
        with patch.object(auth_system.db, 'get_user_by_email',
return_value=None):
            with pytest.raises(AuthenticationError) as exc_
info:
                auth_system.login("nonexistent@example.com",
"any_password")
            assert "Invalid credentials" in str(exc_info.
value)

    def test_login_wrong_password(self, auth_system):
        """Test login with correct email but wrong password"""
        mock_user = Mock()
        mock_user.email = "user@example.com"

        with patch.object(auth_system.db, 'get_user_by_email',
return_value=mock_user):
            with patch.object(auth_system, '_verify_password',
return_value=False):
                with pytest.raises(AuthenticationError) as
exc_info:
                    auth_system.login("user@example.com",
"wrong_password")
                assert "Invalid credentials" in str(exc_info.
 value)

    # JWT Token Tests
    def test_generate_access_token(self, auth_system):
        """Test JWT access token generation"""
        user_id = 1
        token = auth_system.generate_access_token(user_id)

        decoded = jwt.decode(token, "test_secret",
 algorithms=["HS256"])
        assert decoded['user_id'] == user_id
        assert decoded['type'] == 'access'
```

```
        assert 'exp' in decoded

    def test_verify_valid_token(self, auth_system):
        """Test verification of valid JWT token"""
        user_id = 1
        token = auth_system.generate_access_token(user_id)

        result = auth_system.verify_token(token)
        assert result['valid'] == True
        assert result['user_id'] == user_id

    def test_verify_expired_token(self, auth_system):
        """Test verification of expired JWT token"""
        # Create token that expires in -1 hour
        payload = {
            'user_id': 1,
            'type': 'access',
            'exp': datetime.utcnow() - timedelta(hours=1)
        }
        expired_token = jwt.encode(payload, "test_secret",
 algorithm="HS256")
        result = auth_system.verify_token(expired_token)
        assert result['valid'] == False
        assert result['error'] == 'Token expired'

    # Password Reset Tests
    def test_request_password_reset(self, auth_system):
        """Test password reset request"""
        mock_user = Mock()
        mock_user.id = 1
        mock_user.email = "user@example.com"

        with patch.object(auth_system.db, 'get_user_by_email',
return_value=mock_user):
            with patch.object(auth_system, '_send_reset_
email') as mock_send:
                result = auth_system.request_password_
reset("user@example.com")
                assert result['success'] == True
                mock_send.assert_called_once()

    def test_reset_password_success(self, auth_system):
        """Test successful password reset with valid token"""
        reset_token = "valid_reset_token"
        new_password = "NewSecurePass123!"

        with patch.object(auth_system, '_verify_reset_token',
return_value={'valid': True, 'user_id': 1}):
            with patch.object(auth_system.db, 'update_
password', return_value=True):
                result = auth_system.reset_password(reset_
token, new_password)
                assert result['success'] == True
                assert result['message'] == "Password reset
successfully"
```

Step 2: Run tests (they should fail)

Bash

```
# Run the tests - they will fail because implementation
doesn't exist yet
pytest tests/test_auth_system.py -v

# Expected output: All tests fail with ImportError or similar
```

Step 3: Generate implementation with Claude Code

Bash

```
claude-code --task "Implement the AuthSystem class to pass
all the existing tests. Focus on security best practices for
password hashing and JWT token management."
```

Claude Code response

Python

```
# auth_system.py
import bcrypt
import jwt
import re
from datetime import datetime, timedelta
from typing import Dict, Optional, Any
import secrets
import smtplib
from email.mime.text import MIMEText

class AuthenticationError(Exception):
    """Raised when authentication fails"""
    pass

class ValidationError(Exception):
    """Raised when input validation fails"""
    pass

class User:
    def __init__(self, id: int, email: str, password_hash:
str):
        self.id = id
        self.email = email
        self.password_hash = password_hash

class AuthSystem:
    def __init__(self, secret_key: str, db_connection,
                 access_token_expire_hours: int = 1,
                 refresh_token_expire_days: int = 30):
        self.secret_key = secret_key
        self.db = db_connection
        self.access_token_expire_hours = access_token_expire_
hours
        self.refresh_token_expire_days = refresh_token_expire_
days

        # Password requirements
        self.min_password_length = 8
        self.password_pattern = r'^(?=.*[a-z])(?=.*[A-Z])
(?=.*\d)(?=.*[@$!%*?&])[A-Za-z\d@$!%*?&]'

    def register(self, email: str, password: str, confirm_
password: str) -> Dict[str, Any]:
        """Register a new user"""
        # Validate input
        self._validate_registration_input(email, password,
confirm_password)
```

```
        # Check if user already exists
        if self.db.user_exists(email):
            raise ValidationError("Email already registered")

        # Hash password
        password_hash = bcrypt.hashpw(password.encode('utf-8'),
bcrypt.gensalt()).decode('utf-8')

        # Create user in database
        user_id = self.db.create_user(email, password_hash)

        return {
            'success': True,
            'user_id': user_id,
            'message': 'User registered successfully'
        }

    def login(self, email: str, password: str) -> Dict[str,
Any]:
        """Authenticate user and return tokens"""
        # Get user from database
        user = self.db.get_user_by_email(email)
        if not user:
            raise AuthenticationError("Invalid credentials")

        # Verify password
        if not self._verify_password(password, user.password_
hash):
            raise AuthenticationError("Invalid credentials")

        # Generate tokens
        access_token = self.generate_access_token(user.id)
        refresh_token = self.generate_refresh_token(user.id)

        return {
            'success': True,
            'access_token': access_token,
            'refresh_token': refresh_token,
            'user': {
                'id': user.id,
                'email': user.email
            }
        }

    def generate_access_token(self, user_id: int) -> str:
        """Generate JWT access token"""
        payload = {
            'user_id': user_id,
            'type': 'access',
            'exp': datetime.utcnow() + timedelta(hours=self.
access_token_expire_hours),
            'iat': datetime.utcnow()
        }
        return jwt.encode(payload, self.secret_key,
algorithm='HS256')

    def generate_refresh_token(self, user_id: int) -> str:
        """Generate JWT refresh token"""
        payload = {
           'user_id': user_id,
            'type': 'refresh',
            'exp': datetime.utcnow() + timedelta(days=self.
refresh_token_expire_days),
            'iat': datetime.utcnow()
        }
        return jwt.encode(payload, self.secret_key,
algorithm='HS256')

    def verify_token(self, token: str) -> Dict[str, Any]:
        """Verify JWT token"""
        try:
            payload = jwt.decode(token, self.secret_key,
algorithms=['HS256'])
            return {
```

```
                'valid': True,
                'user_id': payload['user_id'],
                'type': payload['type']
            }
        except jwt.ExpiredSignatureError:
            return {'valid': False, 'error': 'Token expired'}
        except jwt.InvalidTokenError:
            return {'valid': False, 'error': 'Invalid token'}

    def request_password_reset(self, email: str) -> Dict[str,
Any]:
        """Request password reset"""
        user = self.db.get_user_by_email(email)
        if not user:
            # Don't reveal if email exists or not
            return {'success': True, 'message': 'Reset
instructions sent if email exists'}

        # Generate reset token
        reset_token = self._generate_reset_token(user.id)

        # Send reset email
        self._send_reset_email(user.email, reset_token)

        return {'success': True, 'message': 'Reset
instructions sent'}

    def reset_password(self, reset_token: str, new_password:
str) -> Dict[str, Any]:
        """Reset user password with token"""
        # Verify reset token
        token_data = self._verify_reset_token(reset_token)
        if not token_data['valid']:
            raise AuthenticationError("Invalid or expired
reset token")

        # Validate new password
        self._validate_password(new_password)

        # Hash new password
        password_hash = bcrypt.hashpw(new_password.
encode('utf-8'), bcrypt.gensalt()).decode('utf-8')

        # Update password in database
        self.db.update_password(token_data['user_id'],
password_hash)

        return {'success': True, 'message': 'Password reset
successfully'}

    def _validate_registration_input(self, email: str,
password: str, confirm_password: str):
        """Validate registration input"""
        if not self._is_valid_email(email):
            raise ValidationError("Invalid email format")

        if password != confirm_password:
            raise ValidationError("Passwords do not match")

        self._validate_password(password)

    def _validate_password(self, password: str):
        """Validate password strength"""
        if len(password) < self.min_password_length:
            raise ValidationError(f"Password must be at least
{self.min_password_length} characters")

        if not re.match(self.password_pattern, password):
            raise ValidationError("Password must contain
uppercase, lowercase, digit, and special character")

    def _is_valid_email(self, email: str) -> bool:
        """Validate email format"""
```

```
        pattern = r'^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.
[a-zA-Z]{2,}$'
        return bool(re.match(pattern, email))

    def _verify_password(self, password: str, password_hash:
 str) -> bool:
        """Verify password against hash"""
        return bcrypt.checkpw(password.encode('utf-8'),
 password_hash.encode('utf-8'))

    def _generate_reset_token(self, user_id: int) -> str:
        """Generate password reset token"""
        payload = {
            'user_id': user_id,
            'type': 'reset',
            'exp': datetime.utcnow() + timedelta(hours=1),  #
 1 hour expiry
            'iat': datetime.utcnow()
        }
        return jwt.encode(payload, self.secret_key,
 algorithm='HS256')

    def _verify_reset_token(self, token: str) -> Dict[str,
 Any]:
        """Verify password reset token"""
        try:
            payload = jwt.decode(token, self.secret_key,
 algorithms=['HS256'])
            if payload['type'] != 'reset':
                return {'valid': False, 'error': 'Invalid
 token type'}
            return {'valid': True, 'user_id': payload['user_
 id']}
        except jwt.ExpiredSignatureError:
            return {'valid': False, 'error': 'Token expired'}
        except jwt.InvalidTokenError:
            return {'valid': False, 'error': 'Invalid token'}
```

```
    def _send_reset_email(self, email: str, reset_token:str):
        """Send password reset email"""
        # In production, implement actual email sending
        # For now, this is a placeholder
        pass
```

Step 4: Run tests again (green phase)

Bash

```
pytest tests/test_auth_system.py -v

# Expected output: All tests pass!
```

Step 5: Refactor and enhance with Claude Code

Bash

```
claude-code --task "Review the authentication system for
security improvements, add rate limiting for login attempts,
and enhance error handling. Maintain all existing test
compatibility"
```

Content-rich prompt example

Define requirements with Claude Code

Bash

```
claude-code --task “Debug authentication failure in user
dashboard - getting 500 errors on login attempts”
```

Environment information

- **Operating system:** macOS Sonoma 14.2.1
- **Node.js version:** v18.17.1
- **npm version:** 9.8.1
- **Framework:** Express.js v4.18.2
- **Database:** PostgreSQL 15.3
- **Key dependencies:**
    - jsonwebtoken: ^9.0.2
    - bcryptjs: ^2.4.3
    - pg: ^8.11.3
    - express-rate-limit: ^6.10.0

Full error information

```
Error: Authentication failed during login process
    at UserController.login (/Users/dev/myapp/controllers/
userController.js:45)
    at Layer.handle [as handle_request] (/Users/dev/myapp/
node_modules/express/lib/router/layer.js:95)
    at next (/Users/dev/myapp/node_modules/express/lib/router/
route.js:144)
    at Route.dispatch (/Users/dev/myapp/node_modules/express/
lib/router/route.js:114)
    at Layer.handle [as handle_request] (/Users/dev/myapp/
node_modules/express/lib/router/layer.js:95)
    at /Users/dev/myapp/node_modules/express/lib/router/index.
js:284
    at Function.process_params (/Users/dev/myapp/node_modules/
express/lib/router/index.js:346)
    at next (/Users/dev/myapp/node_modules/express/lib/router/
index.js:280)
    at /Users/dev/myapp/middleware/auth.js:23
    at Layer.handle [as handle_request] (/Users/dev/myapp/
node_modules/express/lib/router/layer.js:95)

Database connection error in logs:
[2024-01-15 14:23:15] ERROR: connection to server at
“localhost” (127.0.0.1), port 5432 failed: FATAL:  password
authentication failed for user “myapp_user”
[2024-01-15 14:23:15] ERROR: Query execution failed - relation
“users” does not exist
```

Browser console error

```
POST http://localhost:3000/api/auth/login 500 (Internal Server
Error)
Uncaught (in promise) Error: Request failed with status code
500
    at createError (createError.js:16:1)
    at settle (settle.js:17:1)
    at XMLHttpRequest.onloadend (xhr.js:66:1)
```

Reproduction steps

1. Start the application: `npm run dev`
2. Navigate to `http://localhost:3000/dashboard`
3. Enter credentials: `testuser@example.com / password123`
4. Click “Sign In” button
5. Observe 500 error in browser network tab

Check terminal logs showing database connection failure

Expected vs. actual behavior

- **Expected:** User successfully logs in, receives JWT token, and redirects to `/dashboard` with 200 status
- **Actual:** Login attempt returns 500 status with “Authentication failed” message, user remains on login page

Relevant code files

controllers/userController.js (Login method causing error):

Javascript

```
const bcrypt = require('bcryptjs');
const jwt = require('jsonwebtoken');
const User = require('../models/User');

exports.login = async (req, res) => {
  try {
    const { email, password } = req.body;

    // Find user by email
    const user = await User.findByEmail(email);
    if (!user) {
      return res.status(401).json({ message: 'Invalid
credentials' });
    }

    // Verify password
    const isValidPassword = await bcrypt.compare(password,
user.password_hash);
    if (!isValidPassword) {
      return res.status(401).json({ message: 'Invalid
credentials' });
    }

    // Generate JWT token - LINE 45 WHERE ERROR OCCURS
    const token = jwt.sign(
      { userId: user.id, email: user.email },
      process.env.JWT_SECRET,
      { expiresIn: '24h' }
    );

    res.json({
     message: 'Login successful',
```

```
      token: token,
      user: { id: user.id, email: user.email }
    });

  } catch (error) {
    console.error('Login error:', error);
    res.status(500).json({ message: 'Authentication failed'
});
  }
};
```

config/database.js (Database configuration)

Javascript

```
const { Pool } = require('pg');

const pool = new Pool({
  user: process.env.DB_USER || 'myapp_user',
  host: process.env.DB_HOST || 'localhost',
  database: process.env.DB_NAME || 'myapp_db',
  password: process.env.DB_PASSWORD || 'defaultpassword',
  port: process.env.DB_PORT || 5432,
});

module.exports = pool;
```

current .env file

```
NODE_ENV=development
PORT=3000
JWT_SECRET=your-secret-key-here
DB_USER=myapp_user
DB_HOST=localhost
DB_NAME=myapp_db
DB_PASSWORD=wrongpassword123
DB_PORT=5432
```

Database schema (users table)

SQL

```sql
CREATE TABLE users (
  id SERIAL PRIMARY KEY,
  email VARCHAR(255) UNIQUE NOT NULL,
  password_hash VARCHAR(255) NOT NULL,
  created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

Visual context

- **Login form UI:** Standard email/password form with “Sign In” button
- **Network tab:** Shows POST request to `/api/auth/login` returning 500 status
- **Console output:** Application starts successfully on port 3000, but database connection fails immediately when login is attempted

Additional context

- This worked fine yesterday before I updated some dependencies
- PostgreSQL service is running (confirmed with `brew services list`)
- Database exists and user table has test data
- Same issue occurs with different user accounts
- Problem started after running npm update this morning

What I’ve tried

1. Restarted PostgreSQL service
2. Verified database credentials manually with `psql`
3. Checked that users table exists and has data
4. Cleared npm cache and reinstalled node_modules
5. Rolled back to previous git commit - issue persists

Please help identify the root cause and provide a fix that addresses both the database connection issue and any potential problems in the authentication flow.

Well-structured prompt example

Comprehensive Claude Code prompt: Building a REST API for user management

Bash

```
claude-code --task "Build a complete REST API for user management with authentication, validation, and comprehensive testing"
```

Project requirements

Tech stack

- **Runtime:** Node.js v18+ with Express.js framework
- **Database:** PostgreSQL with raw SQL queries (no ORM)
- **Authentication:** JWT tokens with refresh token support
- **Validation:** Input validation and sanitization
- **Testing:** Jest with supertest for API testing
- **Security:** Rate limiting, password hashing (bcrypt), CORS
- **Documentation:** OpenAPI/Swagger specification

Functional requirements

Core user operations

- **Runtime:** Node.js v18+ with Express.js framework
- **Database:** PostgreSQL with raw SQL queries (no ORM)
- **Authentication:** JWT tokens with refresh token support
- **Validation:** Input validation and sanitization
- **Testing:** Jest with supertest for API testing
- **Security:** Rate limiting, password hashing (bcrypt), CORS
- **Documentation:** OpenAPI/Swagger specification

Expected API endpoints

```
POST   /api/auth/register          - Register new user
POST   /api/auth/login             - User login
POST   /api/auth/logout            - User logout
POST   /api/auth/refresh           - Refresh JWT token
POST   /api/auth/forgot-password   - Request password reset
POST   /api/auth/reset-password    - Reset password with token

GET    /api/users/profile          - Get current user profile
PUT    /api/users/profile          - Update current user
profile
DELETE /api/users/profile          - Delete current user
account
PUT    /api/users/change-password  - Change user password

GET    /api/admin/users            - List all users (admin
only)
```

```
GET    /api/admin/users/:id        - Get specific user (admin
only)
PUT    /api/admin/users/:id/status - Update user status (admin
only)
DELETE /api/admin/users/:id        - Delete user (admin only)
```

Data model requirements

Javascript

```javascript
// User entity should include:
{
  id: "UUID primary key",
  email: "unique, validated email address",
  password_hash: "bcrypt hashed password",
  first_name: "required string, 2-50 characters",
  last_name: "required string, 2-50 characters",
  role: "enum: 'user' | 'admin', default 'user'",
  status: "enum: 'active' | 'inactive' | 'suspended', default
'active'",
  email_verified: "boolean, default false",
  last_login: "timestamp, nullable",
  created_at: "timestamp, auto-generated",
  updated_at: "timestamp, auto-updated"
}
```

Security requirements

- Password strength validation (min 8 chars, uppercase, lowercase, number, special char)
- Rate limiting: 5 login attempts per 15 minutes per IP
- JWT tokens expire after 1 hour, refresh tokens after 7 days
- Input sanitization against XSS and SQL injection
- CORS configuration for frontend integration
- Request logging for security auditing

Validation requirements

- Email format validation with proper regex
- Required field validation for all inputs
- Data type validation and length constraints
- Duplicate email prevention
- Password confirmation matching during registration

Error handling

- Consistent error response format across all endpoints
- Appropriate HTTP status codes (200, 201, 400, 401, 403, 404, 409, 500)
- Detailed error messages for development, generic for production
- Request validation errors with field-specific messages

Expected response formats

Javascript

```
// Success response format:
{
  "success": true,
  "data": {...},
  "message": "Operation completed successfully"
}

// Error response format:
{
  "success": false,
  "error": {
    "code": "VALIDATION_ERROR",
    "message": "Invalid input provided",
    "details": [
      {
        "field": "email",
        "message": "Email format is invalid"
      }
    ]
  }
}
```

Database schema: Create PostgreSQL tables with proper indexing, constraints, and relationships. Include migration scripts for easy setup.

Testing requirements:

- Unit tests for all service functions
- Integration tests for all API endpoints
- Test data factories for consistent test setup
- Positive and negative test cases
- Authentication and authorization test scenarios
- Minimum 90% code coverage

Additional features

- Environment-based configuration (development, staging, production)
- Request/response logging middleware
- Health check endpoint (`GET /api/health`)
- API versioning support (`/api/v1/...`)
- Pagination for user listing endpoints
- Basic search functionality for admin user management

Project structure

Organize code with clean architecture:

```
/src
  /controllers    - Request handling logic
  /services       - Business logic layer
  /models         - Data access layer
  /middleware     - Custom middleware (auth, validation, etc.)
  /routes         - API route definitions
  /utils          - Helper functions
  /config         - Configuration files
  /validators     - Input validation schemas
/tests           - Test files mirroring src structure
/docs            - API documentation
/migrations      - Database schema migrations
```

Deliverables Expected

1. Complete Express.js application with all endpoints implemented
2. PostgreSQL database schema with setup scripts
3. Comprehensive test suite with high coverage
4. API documentation (Swagger/OpenAPI spec)
5. README with setup instructions and API usage examples
6. Docker configuration for easy deployment
7. Example environment configuration files
8. Postman collection for API testing

Performance Considerations

- Database connection pooling
- Efficient query patterns with proper indexing
- Response caching where appropriate
- Request timeout handling
- Memory usage optimization

Development Standards

- ESLint configuration with consistent code style
- Proper error handling throughout the application
- Comprehensive logging for debugging
- Clear code comments and documentation
- Git-ready project with proper .gitignore

Please implement this as a production-ready API that follows Node.js and REST API best practices, with emphasis on security, maintainability, and comprehensive testing.

Claude

https://www.anthropic.com/solutions/coding