The game has changed.

AI coding assistants are the new normal.

Building apps today means working with AI coding assistants like Claude, GitHub Copilot, and Cursor. This "vibecoding" approach can get your MVP or POC running in days instead of weeks, but it comes with security trade-offs.

Let's be real: when you're racing to build a prototype or MVP, perfect security isn't your first priority. But major security fails can kill your project just as dead as missing the market window.

We've seen countless founders caught in this trap - they build fast, launch their product, get traction, and then suddenly find themselves dealing with security issues they could have prevented with just a little forethought. The good news? You don't have to choose between speed and security. With the right approach, AI can help you build both quickly and securely.

Not only have we seen this approach, we've lived it. Adversis has built several rapid prototypes and rushed them to market, only we are already security practitioners, so we focused on building secure apps from the start.

This guide shows you how to build secure-enough applications with AI coding assistants without slowing down your development speed. Think of it as your practical playbook for the new era of AI-assisted development.

Principles for AI-Assisted Development

CHOOSE TECHNOLOGY THAT ELIMINATES ENTIRE CATEGORIES OF VULNERABILITIES

Serverless over servers - Use services like Vercel, Netlify or AWS Lambda instead of managing your own servers. Serverless functions are ephemeral, automatically patched, and scale to zero when not in use. This means no outdated software, no unpatched vulnerabilities, and no persistent environment to attack.

NoSQL over SQL (for prototyping) - Document databases like MongoDB or Firestore are often more forgiving for MVPs. They typically have simpler query patterns that are less susceptible to injection attacks. You can query by document structure rather than writing complex SQL that might introduce vulnerabilities.

Managed services over self-hosted - Use Auth0 for authentication, Cloudinary for file storage, Stripe for payments, etc. These companies have entire security teams dedicated to what would otherwise be a side project for you. Their security expertise vastly outweighs what you can implement quickly.

Edge functions over long-running services - Edge functions run close to users and are destroyed after each execution. This creates a naturally secure boundary because there's no persistent state to attack. Cloudflare Workers, Vercel Edge Functions, and Netlify Edge Functions all provide this benefit.

Static sites over dynamic rendering - Static site generation (SSG) produces plain HTML/CSS/JS files that don't require server-side execution to render. This dramatically reduces the attack surface since there's no dynamic code execution on each request. Use Next.js, Astro, or similar frameworks with static generation when possible.

Mature frameworks over custom code - Frameworks like Next.js, Remix, or SvelteKit have been security-hardened by thousands of developers. They handle things like sanitizing inputs, preventing XSS, and managing CSRF tokens automatically. Use their built-in features instead of rolling your own.

DATA MINIMIZATION - DONT COLLECT WHAT YOU DON'T NEED

Store only essential data - For each piece of user data, ask "Do we actually need this?" If you don't collect it, you can't leak it. For an MVP, start with the absolute minimum and add fields only when proven necessary.

Minimize sensitive information - Avoid collecting sensitive data like full addresses when approximate location would work, or full birth dates when age ranges would suffice. Consider if you really need a user's real name or if a username would work.

Use tokenization for sensitive fields - When you must handle sensitive data (like payment information), use tokenization services. Stripe, for example, gives you tokens to store instead of actual credit card numbers. You get the functionality without the security risk.

Set aggressive data retention policies - Automatically delete data that's no longer needed. Even for an MVP, implement simple time-based cleanup jobs. Many breaches expose data that should have been deleted years ago.

Use one-way hashing where possible - For data you need to verify but never need to retrieve in its original form (like passwords), store one-way hashed versions with strong algorithms like bcrypt or argon2.

MODULAR, SIMPLE DESIGN

Small, focused components - Each component should do one thing well. This makes security reviews easier and contains potential vulnerabilities. For example, separate your authentication logic from your business logic.

Clear interfaces between modules - Well-defined interfaces limit how components can interact, reducing unexpected behavior. Document what each function expects and returns, and validate at the boundaries.

Single responsibility functions - Functions should do one thing only. This makes them easier to test, reason about, and secure. A function named `validateAndSaveUser` is probably doing too much.

Explicit dependencies - Make dependencies clear and intentional. Avoid global state and implicit dependencies that make code hard to understand and secure.

Minimal state management - State is a common source of security bugs. Keep it simple and contained. Use immutable patterns where possible and centralize state management in a well-tested library.

LEVERAGE MANAGED SECURITY

Auth providers - Services like Auth0, Clerk, Supabase Auth, or NextAuth handle the complex parts of authentication for you. They manage password hashing, session security, multi-factor authentication, and security monitoring with minimal setup.

Managed databases with built-in security - Services like Supabase, Firebase, PlanetScale, or MongoDB Atlas include security features like connection encryption, automatic backups, access controls, and audit logging out of the box.

CDNs with WAF protection - Content Delivery Networks like Cloudflare or AWS CloudFront include Web Application Firewalls that block common attacks automatically. They identify and stop malicious traffic before it ever reaches your application.

API gateways with rate limiting - Services like Kong, AWS API Gateway, or simple middleware libraries automatically limit request rates to prevent abuse. They ensure no single user can overwhelm your system with too many requests.

Managed secret storage - Use services like AWS Secrets Manager, GitHub Secrets, or Vercel Environment Variables to store sensitive configuration securely. These services encrypt your secrets at rest and limit access to authorized services.

> **“Assume hostile users will attempt to abuse these features. Include appropriate protections.”**

Secure Alternatives to High-Risk Features

Some features are inherently higher risk. Consider just.. not using them.

When building MVPs, traditional username/password authentication brings complexity and security risks. Consider these safer alternatives that provide similar functionality with much less security overhead:

Magic links instead of passwords - Send one-time login links to users' email addresses instead of storing passwords. Services like Magic.link, Supabase Auth, or NextAuth make this simple to implement. Users click a link in their email to sign in, eliminating password storage entirely.

OAuth providers over custom auth - Let users sign in with Google, GitHub, or other OAuth providers. These companies handle the hard parts of authentication, and you just verify the tokens they provide. This offloads security complexity to experts.

Session-based auth over JWT for MVPs - For simpler applications, session-based authentication is often easier to implement securely. JWT management introduces complexities around token revocation and expiration that session-based auth handles more naturally.

Passwordless authentication - Consider SMS codes, WebAuthn, or biometric authentication when possible. These methods avoid password storage altogether and provide better user experience.

AUTHENTICATION

Handling payments is one of the riskiest parts of any application. These alternatives can dramatically reduce your security burden while still providing great customer experiences:

Stripe Checkout over custom payment forms - Use Stripe's hosted checkout page instead of building your own payment form. Stripe handles PCI compliance, fraud detection, and secure card storage. Your application never touches the credit card data.

Payment links for simple use cases - For basic payments, generate payment links with services like Stripe Payment Links or PayPal. These require zero coding for basic scenarios and are fully secured by the provider.

Subscription management services - Use services like Stripe Billing, Chargebee, or Paddle to manage subscriptions. They handle all the complex billing logic, proration, invoicing, and payment security.

PAYMENT PROCESSING

Data storage and access patterns can introduce subtle security vulnerabilities. These alternative approaches can eliminate entire classes of risks:

UUIDs instead of sequential IDs - Use UUIDs or other non-sequential identifiers for database records. This prevents enumeration attacks where attackers can guess valid IDs by incrementing numbers.

Composite keys over simple keys - Use composite keys that combine multiple fields for more security. This makes it harder for attackers to guess valid identifiers.

Truncated or tokenized data - Store only the portions of sensitive data you actually need. For example, store only the last four digits of credit cards or phone numbers when possible.

DATA

File uploads and storage create significant security challenges. These alternatives simplify security while maintaining functionality:

Cloud storage over local storage - Use AWS S3, Cloudinary, or similar services instead of storing files locally. These services handle security, scaling, and backups automatically.

Pre-signed URLs over direct access - Generate time-limited, pre-signed URLs for file uploads and downloads instead of directly connecting users to your storage. This provides fine-grained access control and prevents unauthorized access.

Content delivery networks - Use CDNs like Cloudflare or Fastly to serve static assets. They provide DDoS protection, caching, and geographic distribution with minimal setup.

FILE HANDLING

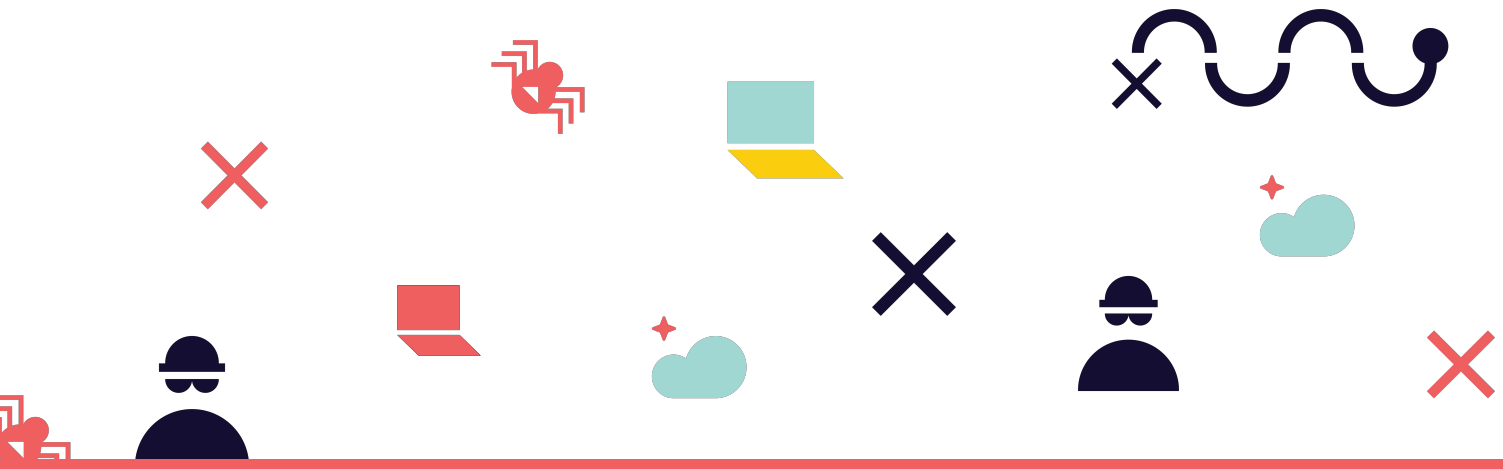
Sending emails, SMS messages, and notifications securely requires specialized infrastructure. These alternatives let you offload that complexity:

Managed email services - Use services like SendGrid, Mailgun, or AWS SES instead of running your own mail server. These services handle deliverability, security, and compliance automatically.

SMS providers - Use Twilio, MessageBird, or similar services for SMS rather than direct carrier integrations. They provide security features and abstractions that simplify secure implementation.

Push notification services - Use Firebase Cloud Messaging, Apple Push Notification Service, or OneSignal through their managed SDKs rather than building custom notification systems.

COMMS



AI Assisted Secure Development

Prompt Engineering FTW

STEP 1: SETUP PROMPT

On day one of your project, have your AI assistant set up security fundamentals.

```
I'm building an MVP for [describe your app]. I need to move fast
but keep reasonable security, assuming malicious users will try to
abuse it. Please:

1. Recommend a serverless architecture that minimizes security
risks
2. Suggest a minimalist tech stack with managed services where
possible
3. Set up a basic project structure with security defaults
4. Create minimal but effective security controls for
authentication and data access
5. Focus on simplicity and maintainability
6. Use only the latest stable versions of all libraries and
frameworks
7. Never store or expose secrets in client-side code
8. Protect against business logic, rate limits, and payment abuse

Tech requirements:
- Need to store [type of data]
- Authentication for [type of users]…
```

STEP 2: CREATE A CONTEXT FILE

Create a simple security context file that your AI can reference.

```
# Project Context

This is an MVP/POC with balanced security requirements.

## Security Priorities
- Use managed services for auth and data storage
- Validate all user inputs
- Use parameterized queries for database access
- Implement basic rate limiting
- Maintain HTTPS throughout
- Follow a "secure by default" approach where it doesn't slow
development
- Never expose secrets in client-side code or API responses
- Always use the latest stable versions of dependencies
- No test/debug endpoints that expose sensitive information

## Tech Stack
- Next.js 13+ app router
- Vercel serverless functions
- Supabase for auth and database
- Simple zod validation
- Environment variables for secrets
- No unmaintained or deprecated libraries

## Dependency Requirements
- React 18.2.0 or newer
- Next.js 13.4.0 or newer
- zod 3.21.0 or newer
- TypeScript 5.0.0 or newer
- Only use actively maintained libraries with recent updates

## Development Approach
- Favor simplicity over complexity
- Use existing libraries over custom implementations
- Focus on modular, reusable components
- Balance security and development speed
- Never return sensitive data to clients
```

claude.md

200%

More vulnerabilities identified in vibe coded apps vs traditional app development.

"You highlighted legitimate concerns... A+ would read again."

- Penetration Testing Client

STEP 3: CREATE CORE TEMPLATES

Have your AI generate templates for common operations

```
Please create these minimal but secure templates for our MVP:

1. A basic API endpoint template with:
   - Simple input validation
   - Authentication check
   - Basic error handling
   - Clean separation of concerns
   - No exposure of secrets or sensitive data

2. A database operation template that:
   - Uses parameterized queries
   - Has proper error handling
   - Follows least privilege
   - Only returns necessary fields (never sensitive data)

3. A form submission handler that:
   - Validates inputs
   - Prevents common attacks
   - Provides good UX for errors
   - Never exposes secrets client-side
```

STEP 4: SEND IT

At this point, most of the security foundation has been put in place. Now, strategic prompting will keep your AI coding assistant on track to follow your guidance.

```
We need a simple file upload
feature for our MVP:

1. Use a managed service (S3/
Cloudinary) rather than storing
files ourselves
2. Implement basic file
validation (size, type)
3. Use signed URLs for uploads
4. Keep the implementation simple
but secure
5. Focus on the happy path but
handle basic error cases

As this is a prototype, we need
something working quickly that
doesn't have major security
holes.
```

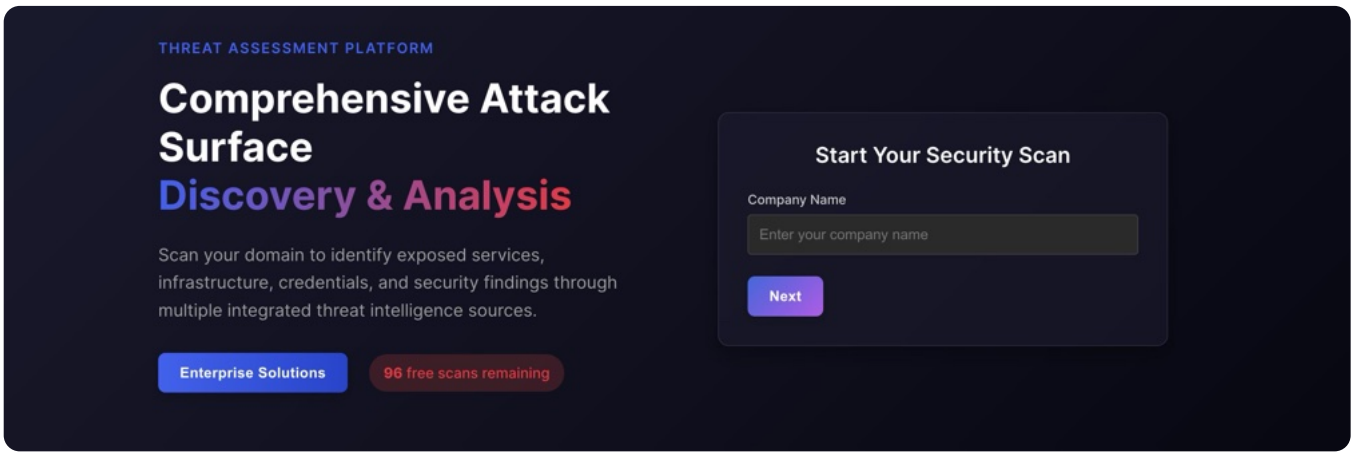
```
Create a basic API for our user
profiles feature:

1. Use our serverless API
template
2. Implement simple validation
with zod
3. Use Supabase for data storage
with parameterized queries
4. Add basic error handling
5. Implement authentication
checks

This is for an MVP, so keep it
simple while addressing
fundamental security needs.
```

CASE STUDY

threatscan.ai

THREATSCAN.AI IS A THREAT INTELLIGENCE PLATFORM BUILT AS AN MVP WITH SECURITY BAKED IN FROM THE START.

Adversis initially developed an AI-powered threat intelligence service as a proof of concept to demonstrate value during sales conversations. We quickly recognized its broader potential and decided to make it publicly available.



Architecture Overview

Token-based access - No traditional authentication system

MongoDB with TTL indexes - Auto-expiring data for scan results

Serverless API - No persistent servers to attack

External API Based Functionality - Minimal server side logic

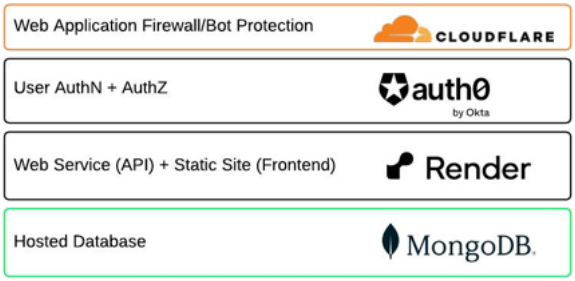
Infrastructure Overview

Render - No persistent server infrastructure

Mongo Atlas - Hosted database in SaaS provider

Stripe - No custom payment processing

GItLab - CICD and source code live in GitLab



CASE STUDY

threatscan.ai

THREATSCAN.AI IS A THREAT INTELLIGENCE PLATFORM BUILT AS AN MVP WITH SECURITY BAKED IN FROM THE START.

During our beta launch, we discovered a issue in our token-based access system that could have cost us $847 in about 2 minutes. Our initial prompt was too basic.

```
# "Create an API endpoint that accepts a token and list of
domains to scan"

app.post('/api/scan', async (req, res) => {
  const { domains } = req.body;
    domains.forEach(domain => scanDomain(domain));
});
```

See the issue? A single user could submit 10,000+ domains, causing massive API costs and service degradation.

```
# "Create an API endpoint for domain scanning with secure
constraints, assuming someone will try to abuse it.
- Implement reasonable per-token rate limiting
- Maximum 100 scans per token per 24 hours
- Each external API call costs $0.10, protect against abuse"
```

Due to size we won't print it here, but now we have daily limits, token checks, and assurance that abuse will be minimized.

Prompt should include an attacker's perspective. Whether you're building a contact form, API endpoint, or data processor, consider the threat model and abuse cases. Add the following to your prompt:

"Assume hostile users will attempt to abuse this feature. Include appropriate protections."

Defense prompting will help reduce software and business logic vulnerabilities in your applications. You don't need to know every possible attack either - just remind the agent that these attacks exist!

Now use these tools to securely MVP and bring in an expert as complexity increases.

You face impossible choices

- Move fast and risk security breaches
- Lock everything down and lose business agility
- Check compliance boxes and still get breached

While Fortune 500s have elite security teams, the rest of the 99% are left hoping.

We're building a world where breaches are no longer the norm and your data stays private.

Adversis was founded by red team veterans who've breached the world's most sophisticated systems, Adversis translates fear, uncertainty, and doubt into confidence, speed, and growth.