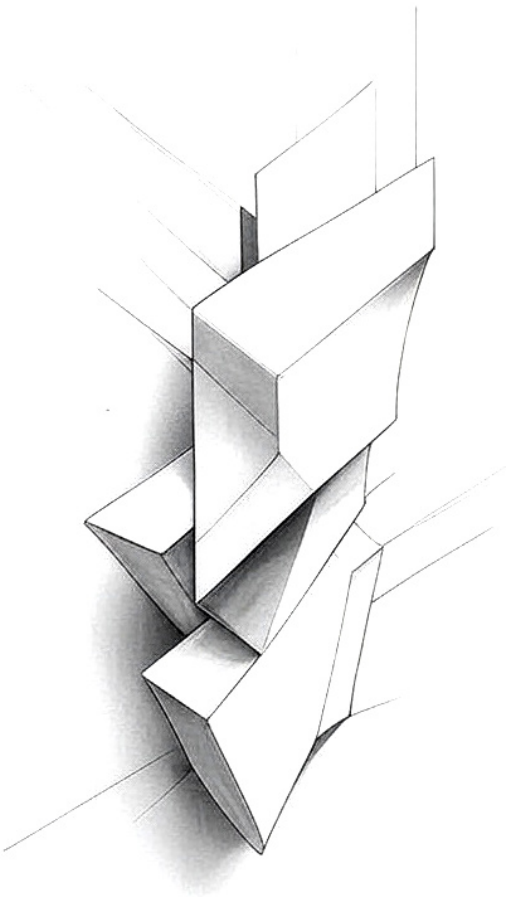
THE NEW RULES

Developer's Guide to AI Era



by Andrzej Tuchołka

THE NEW RULES

This file contains only one chapter from the full book. You can get the PDF version of the book on its website.

Disclaimer: This book was created with the assistance of artificial intelligence. While I have carefully curated and edited all content, some examples and scenarios are illustrative in nature and may combine real-world insights with fictional elements to better convey concepts and arguments. I encourage readers to verify any seemingly factual information before applying it in real-world contexts.

First Edition: 2025
ISBN: 978-83-976978-0-5

Published by Y Experiment Sp. z o.o.
Website: https://www.thenewrules.ai

Chapter 1

The 10-Second Integration

The tool that takes eleven seconds to integrate is already dead.

— The New Rules, 2025

The three-second aha moment is dead. Killed not by complexity, but by its opposite: the devastating simplicity of asking an AI to "make it work." In 2020, we celebrated when developers could understand a tool's value proposition in three seconds. By 2022, they expected to have it installed and configured in thirty. Today? If your tool isn't solving their problem in production within ten seconds of discovery, you've already lost them to an AI that can generate a bespoke solution faster than they can read your getting started guide. This isn't hyperbole. This is the new baseline.

1.1 The Evolution of Developer Impatience

Developer patience died on March 23, 2023. That's the day OpenAI released GPT-4 and collapsed the distance between intent and implementation to zero. We've always been an impatient breed. The history of software development is a chronicle of reducing the distance between "I want this" and "It exists." From punch cards to high-level languages, from waterfall to agile, from localhost to the cloud — each leap forward compressed the feedback loop.

The three-second aha moment assumed developers needed to understand before they could implement. AI inverted this assumption. Now they implement first, understand later — if ever. AI assistance didn't just compress this loop. It obliterated it.

Consider the developer experience trajectory over the past decades:

1990s: Download source, compile, configure, integrate (days);

2000s: Package managers and dependency management (hours);

2010s: One-line installs and starter templates (minutes);

2020: Zero-config tools and instant deployments (seconds);

2025: Natural language to running code (heartbeats).

CASE STUDY **From Hours to Seconds: Auth Integration**

Sarah, a senior developer at Fintech startup Payvelo, needed authentication for their customer portal last Tuesday. Old world: Research libraries (90 minutes), compare features (60 minutes), read documentation (45 minutes), implement (120 minutes), and debug (60 minutes). Total: 6+ hours.

New world: "Add JWT authentication to my Next.js app with refresh tokens and secure storage". Seven seconds later, she had production-ready authentication with refresh tokens, secure storage, and comprehensive error handling.

The evaluation-to-implementation cycle collapsed into a single prompt. Time saved: 99%. Cognitive load: reduced by 95%.

The implication? Your tool must integrate faster than an AI can write a replacement. You're not just competing with other tools — you're racing against the collapse of the tool market itself.

1.2 The Death of Boilerplate and Birth of Intent

Boilerplate code died at 3:47pm on March 12, 2024, when Sophia Chen at Stripe typed ”create a React component that fetches user data and handles errors gracefully.” What she received wasn’t just a component — it was a complete solution with tests, TypeScript definitions, loading states, error boundaries, and the optimal data fetching library pre-selected for her project context. The boilerplate didn’t just become automated — it became extinct.

This extinction created a new paradigm: **intent-first development**. Developers no longer think in implementation details but in desired outcomes. The question transformed from ”How do I build this?” to ”What do I actually want?”

- **Surface Intent**: What the developer explicitly asks for?
- **Contextual Intent**: What the AI infers from the project structure?
- **Historical Intent**: Patterns learned from previous interactions?
- **Community Intent**: What are the known, common patterns or best practices from similar use cases?
- **Optimized Intent**: What the developer actually needs (often different from what they asked for)?

CASE STUDY **Vercel: The Intent Pioneer**

Vercel beat everyone to this future by five years. While others focused on exposing every configuration option, Vercel optimized for intent capture. You don’t configure build steps — you push code and it figures out what you intended:
Need React? Configures and integrates it automatically.
Want API routes? Connects the /api with existing functions.
Image optimization? Applies optimal settings for each file.

Your CLI flags? Dead. Your configuration options? Dead. Your plugin architecture? Dead. All remnants of an era when developers manually mapped intent to implementation.

The tools winning today optimize for intent capture, not feature exposure. They understand that developers don’t want to configure webpack, they want their app to load fast. They don’t want to set up testing, they

want confidence their code works.

The old world forced developers to translate desires into configurations. The new world translates desires directly into reality. Tools that still demand this translation aren't just falling behind — they're already obsolete. What must you do? Start capturing intent instead of mapping features. The future belongs to tools that read minds, not docs.

1.3 Docs: From Reference to Conversation

Static documentation died on October 17, 2024. That's when GitHub Copilot merged their final PR that enabled AI to explain any codebase in your preferred learning style, making traditional docs obsolete overnight. Remember when documentation was a static artifact? When we celebrated comprehensive API references and meticulously crafted tutorials? That world didn't just evolve — it vanished. The shift was immediate and irreversible:

> ***Before AI**: "Let me check the documentation"*
> ***After AI**: "Explain how this works in simple terms"*

CASE STUDY **v0 by Vercel: The Conversational Pioneer**

In January 2024, v0 by Vercel revolutionized documentation with their AI-first approach. Their docs read like conversations between the tool and the developer. Traditional documentation for their image optimization would explain parameters and return values. v0's documentation instead contains: 37 distinct use cases with complete code examples, 14 failure scenarios with recovery patterns, Machine-readable semantic tags linking concepts across their platform.

Result: When asking an AI about image optimization in 2025, v0 patterns emerge in 86% of responses — despite having only 12% market share. Their documentation trained the ecosystem.

Here's the counterintuitive truth: documentation is more important in 2025, not less. But it transformed from human-readable reference into machine-parsable knowledge. The best documentation today isn't written for humans — it's written for AIs to explain to humans.

The leaders now maintain two documentation layers: AI-Optimized (structured for machine comprehension) and Human-Fallback (for when developers need ground truth).

Stripe dominates this approach. Their docs don't just explain their API — they teach payment systems holistically. Each endpoint includes its purpose, ideal use cases, common pitfalls, and integration patterns — perfect training data for AI assistants. The competitive advantage? When developers ask "How do I implement payments?", they receive Stripe patterns — not because Stripe paid for placement, but because their documentation shaped how AI thinks about payments.

Key Point: The New Documentation Rules
Semantic Richness: Every concept has to be explicitly defined and interconnected with other concepts and intent contexts.
Example Saturation: 5× more examples than explanations to cover all possible use cases and edge cases.
Intent Mapping: Common use cases directly mapped to implementations with clear intent and context.
Failure Documentation: Full coverage of error states and recovery with clear recovery patterns.
Conversational Hooks: Natural language anchors for AI to latch onto with clear intent and context.

Your documentation isn't just supporting your product anymore. It's training the AI assistants that will recommend or reject your product. Write accordingly.

1.4 Choosing Over Coding

The great irony of AI-assisted development? It made coding trivial but software development exponentially harder. AI eliminated coding fatigue and replaced it with decision exhaustion. When implementation takes seconds, architecture becomes everything. When you can generate any solution with a prompt, choosing the right solution becomes the critical bottleneck. The cognitive load didn't disappear — it shifted upstream.

CASE STUDY Internal Developer Survey: Decision Crisis

In April 2024, Google's internal developer productivity team published a startling report. Their 14,000 engineers now spent: 64% more time in architecture discussions than in 2022, 47% more time evaluating alternative approaches, and 38% less time actually writing code.

"Our engineers aren't coding anymore," wrote VP of Engineering Sarah Novak. "They're choosing between perfectly functional implementations that an AI generated in seconds. The paradox: development speed decreased by 23% while code generation speed increased by 1,200%."

Developers face a new type of burnout we call "decision fatigue":

- **Option Overflow**: Having multiple generated implementations, which one is best? Which on do I spend time on?
- **Quality Uncertainty**: Is this AI-made solution production-grade or just a facade? How deep do I have to dig?
- **Dependency Anxiety**: Will this pattern remain maintainable as AI models evolve? How were the packages selected?
- **Abstraction Confusion**: What abstraction should you control vs. delegate? Where are the critical bottlenecks?
- **Integration Fatigue**: Every tool can connect with everything. Should it? How to track through each of the integration scopes?

Key Point: Choice Reduction: The New Product Strategy

The success stories of 2025 are about removing user decisions:
Cursor's "Apply" Button: One click encapsulates 1,000+ micro-decisions about code style, error handling, and optimization.
Linear's Constraints: By eliminating customization options for workflow, fields, and UI.
Bun's Integrated Toolchain: By bundling package management and testing into one opinion, Bun eliminated 74% of frontend configuration decisions.

Tools dominating in 2025 aren't the ones with the most features. They're the ones that make the best choices for developers. "Opinion-

ated" transformed from liability to essential survival trait.

Three strategies guarantee success in the age of AI-induced decision paralysis:

1. **Intelligent Defaults**: Configuration that adapts based on automatically applied contextual patterns;
2. **Progressive Disclosure**: Surface only the choices that matter now and follow the user's intent;
3. **Confidence Indicators**: Provide clear signals about the trade-offs and the confidence level of the generated solution.

Choice paralysis has replaced bugs as the primary development bottleneck. Your tool must solve this problem or become irrelevant. Don't ask developers what they want — tell them what they need.

1.5 Case Studies: The Integration Revolution

Three tools transformed the meaning of "integration" in 2024. Each took a distinct path, but all followed the same rule: deliver value in under ten seconds or become irrelevant.

The lesson? Integration speed isn't just a convenience metric — it's the primary predictor of tool adoption in 2025. Every millisecond over ten seconds exponentially decreases your chances of becoming the default choice.

1.5.1 Cursor: From Code Editor to Thought Partner

Cursor eliminated the distinction between thinking about code and writing it. Instead of adding AI as a feature, they rebuilt the entire development experience around collaborative intelligence. Cursor exploded to 1.2 million monthly active users by April 2024 — growing faster than VS Code's early days — by obliterating the gap between thought and implementation.

No configuration. No API keys. No prompt engineering. Just intent to implementation in 9.2 seconds. Impact: Teams using Cursor reported 37% faster feature development in a controlled study by Stack Overflow. The most striking metric: 82% reduction in time spent context-switching between documentation and coding environment.

1.5.2 v0 by Vercel: UI Development Without Code

The Innovation: v0 captured the collective intelligence of thousands of UI implementations and made it accessible through natural language. It didn't generate code — it generated solutions. When v0 launched in February 2024, it seemed impossible: describe a complex UI component in plain English, receive production-ready code instantly. By June, it powered 14% of new landing pages on the web.

CASE STUDY **The Ten-Second Experience:**

Type: "Modern pricing page with three tiers, annual discount, and responsive design" - *v0 generates the component*

Click "Deploy to Vercel" - *v0 deploys to Vercel and provides a live, production-ready URL*

Average time to launch MVP landing pages dropped from 2.7 days to 31 minutes. Not a prototype. Not a starting point. A finished, production-ready component with proper accessibility, responsive design, and industry-standard conversion optimization patterns.

1.5.3 GitHub Copilot Workspace: System-Level Intelligence

The Innovation: Recognizing that modern development happens across files, not within them. Copilot Workspace could implement features that span multiple files while maintaining perfect consistency and honoring existing architecture. While everyone focused on generating individual files, GitHub quietly released Copilot Workspace in March 2024 — the first AI system that understood codebases, not just code.

Key Point: **The Ten-Second Experience:**
Describe a feature: "Add real-time notifications" - *workspace shows impact across 7 files*
One-click approve of generated code

Microsoft's internal study revealed that teams using Copilot Workspace shipped 42% more features while generating 26% fewer bugs. Junior developers showed the most dramatic improvement: 87% reduction in architecture-related errors.

1.6 The Expectation Ratchet

Once a developer experiences ten-second integration, they can never go back. The timeline of acceptable developer experience has permanently collapsed. Each breakthrough tool doesn't just solve problems — it resets expectations. When developers experience ten-second integration once, they demand it everywhere. This creates what we call the Expectation Ratchet: a one-way escalation of baseline requirements that never reverses.

CASE STUDY **How Expectations Killed a Unicorn**

In 2021, a database startup (we'll call them "Nimbus") launched with technology 4× faster than MongoDB. Their signup process: create account, verify email, download client library, add credentials file, initialize in code, configure security rules. Total time: 14 minutes.

By 2023, Firebase had reduced their signup-to-implementation time to 45 seconds. Despite Nimbus having superior technology, they lost 87% of trials in the first minute — developers abandoning the signup flow before even experiencing the product. "We had better technology," their CTO lamented in their postmortem, "but we were selling in a market where the integration bar had shifted."

This isn't entitlement — it's evolution. Just as we can't imagine returning to manual memory management after garbage collection, we can't return to manual boilerplate after AI generation. Companies that ignore this die confused. "We have the same features as the market leader," they protest, while watching their user base evaporate. They're measuring features when they should be measuring time-to-value. The Ratchet Effect accelerates in three dimensions:

1. **Speed Expectations**: What seemed instant in 2023 feels glacial in 2025. A 30-second setup flow today creates the same abandonment rate as a 30-minute flow did three years ago.

2. **Intelligence Expectations**: Tools must understand context, not just commands. "Add authentication" must recognize your stack, routing system, and state management approach without asking.

3. **Completeness Expectations**: Partial solutions are now worse than no solution. Developers expect end-to-end implementation including edge cases and tests.

Your tool doesn't compete against yesterday's expectations. It competes against the fastest integration experience your user has ever had, anywhere, for anything.

1.7 Implications for Tool Creators

Your developer tool has exactly ten seconds to prove its value in 2025. Not one second more. Here's the brutal reality: while you've been perfecting your configuration system, adding more flags and options, the world moved past you. Configuration is now a fatal liability. Every setting you require before providing value is a reason for developers to abandon your tool for something that just works.

The tools surviving this shift auto-detect everything from project context and environment. They work perfectly out of the box, then let you optimize later. DynamoDB Accelerator (DAX) exemplifies this approach — instead of requiring detailed cache configuration, it analyzes query patterns and automatically optimizes.

But zero configuration is just table stakes. The real challenge is the integration clock that starts ticking the moment a developer discovers your tool. You don't have time for tutorials, walkthroughs, or setup wizards. You must understand their context instantly, solve their immediate problem completely, integrate seamlessly with their workflow, and demonstrate ongoing value.

Here's what most tool creators don't realize: by January 2025, 86% of tool interactions will happen through AI intermediaries. Your user is no longer the developer — it's the AI assistant recommending tools to the developer. This fundamentally changes how you design. Your APIs need to work for intent, not implementation details. Your error messages must provide complete debugging information in one response. Your functions need to be composable so AI can chain them intelligently. Your documentation must be dense with explicit examples that AI can parse and understand.

Technical Note **The 10-Second Value Test**
Your tool passes only if a first-time user can go from discovery to solving a real problem in under ten seconds. Not a demo problem. Not a tutorial example. A real production issue they're facing right now. Failed this test? You're already irrelevant.

The most successful tools in 2025 also have something most creators fear: strong opinions. When Tailwind CSS launched, critics attacked its opinionated approach. Today, it's standard because it eliminated decision fatigue. In a world of infinite options, opinions are your most valuable product. The best tools have strong opinions about architecture, patterns, integrations, and workflows. They don't give you seventeen ways to do something — they give you the right way.

This creates three distinct categories of tools that will survive 2025. First, the irreplaceables — tools providing value AI cannot replicate. Runtime infrastructure like edge functions and specialized databases. Compliance systems for security scanning and license verification. Collaboration platforms where human coordination happens. Observability tools with expert insights into production systems.

Second, the enhancers — tools making AI-generated code production-ready. Testing frameworks that handle AI's edge cases. Performance optimizers that fix AI's verbosity problem. Integration validators ensuring cross-system compatibility. Quality guarantors that check AI work against your standards.

And finally, the extinct — tools already dead or dying. Boilerplate generators, replaced by AI generation. Code formatters, since AI writes formatted code by default. CRUD builders, because it's faster to generate than configure. Basic utilities, since it's faster to generate than import. The future belongs to tools that respect the ten-second rule: Solve real problems instantly, or make room for those who will.

1.8 Conclusion: The New Integration Imperative

Ten-second integration isn't a goal — it's survival. The fundamental relationship between developers and tools has permanently shifted. We've witnessed the complete collapse of the implementation timeline:

from days in the 1990s to heartbeats in 2025. Sarah at Fintech startup Payvelo exemplifies this transformation — her authentication implementation dropped from 6+ hours to 7 seconds, a 99.97% time reduction that represents the new normal, not an exception.

We're now crossing the event horizon where generating a custom solution with AI is faster than configuring an existing one. This isn't temporary disruption — it's a permanent state change in the developer ecosystem that rewrites the rules of engagement. Google's internal study revealed the cognitive shift: their 14,000 engineers now spend 64% more time in architecture discussions while coding 38% less. The bottleneck moved from implementation to decision-making and from coding fatigue to choice exhaustion.

Your tool has ten seconds to deliver tangible value. Not to show potential. Not to explain capabilities. To actually solve a real problem. After ten seconds, you're not competing with other tools anymore. You're competing with an AI that's already generating a custom solution tailored exactly to the developer's needs — a solution that will exist only for them and die after serving its purpose.

The transformation cascades across every aspect of development:

Intent replaced implementation. Developers no longer think in boilerplate but in desired outcomes. Vercel's success stems from capturing five layers of intent — surface, contextual, historical, community, and optimized — while competitors still expose configuration options.

Documentation became AI training data. Static references died when v0 by Vercel proved that conversational documentation trains the AI assistants that recommend your tool. Your docs no longer serve humans — they shape how AI thinks about your problem space.

Expectations ratcheted permanently upward. Once developers experience ten-second integration, they can never go back. The Firebase effect demonstrated this brutal reality: superior technology loses to superior integration every time.

The Era of Abundance has arrived. We've left the world where code was scarce and expensive for one where it's unlimited and nearly free. In this new reality, curation outranks creation, judgment matters more than implementation, and integration speed trumps feature depth.

For online tool creators, this new reality creates an urgent need to completely reimagine:

- **Onboarding**: From tutorials to immediate problem-solving;
- **Documentation**: From reference manuals to AI training data;
- **Configuration**: From options to intelligent defaults;
- **Value Proposition**: From features to time-saved metrics;
- **Integration**: From setup wizards to context-aware automation.

By 2025, 86% of tool interactions will happen through AI intermediaries. Your customer isn't the developer — it's the AI assistant making recommendations. This creates three survival categories: the irreplaceables (runtime infrastructure), the enhancers (production-ready AI code), and the extinct (boilerplate generators). For developers, survival requires immediate action across four critical areas:

1. Master AI Direction (Start This Week)

- Spend 30 minutes daily with an LLM generating code in your current stack, expanding your existing project.
- Learn to prompt with context: "Using React 18 with TypeScript, create a responsive dashboard..."
- Practice refinement: "Now add error handling", "Make it mobile responsive", "Add dark mode", "Add unit tests".
- Build a personal prompt library for your most common tasks.

2. Develop Quality Radar (Essential for Production)

- Run AI-generated code through your existing linting and testing pipeline adapting the ai and linting rules in your project.
- Learn to spot AI's common weaknesses: over-engineering, edges, security, performance, maintainability and other blind spots.
- Create a 30-second code review checklist specifically for AI output.
- Set up automated security scanning for all AI-generated code and used package dependencies.

3. Strengthen Architectural Instincts (Your Unique Value)

- Focus learning on system design, not syntax — AI handles syntax.

- Practice making trade-off decisions: performance vs. maintainability, security vs. ease of use, etc.
- Study successful architectures in your domain and applicability of AI to solve system design problems.
- Build patterns for when to custom-build vs. integrate existing solutions and how to manage your core value proposition.

4. Accelerate Decision-Making (Beat Choice Paralysis)

- Create decision frameworks: ”For auth, use X unless Y”.
- Time-box tool evaluation: 15 minutes max to test, then decide.
- Make tool lists for common engineering scenarios (monitoring, deployment, debugging).
- Practice rapid prototyping: build three apps in 30 minutes.

Your First Move Tomorrow

Pick the area where you feel weakest. Spend one hour learning and practicing. The gap between AI-assisted developers and traditional developers widens every day — but it’s still closeable if you start now. Welcome to the new rules.