The Autonomous Development Stack Playbook for Cursor AI

The Master .cursorrules Configuration

Save this as your project's `.cursorrules` file for maximum autonomous development:

yaml

```
# ============================================
# AUTONOMOUS DEVELOPMENT STACK - MASTER CONFIG
# ============================================

# PRIORITY 1: AUTONOMOUS OPERATION CORE
DO NOT GIVE ME HIGH LEVEL SHIT - give actual code or implementation
Never use placeholders like "// ... rest of implementation" or "// Add error handling here"
ALWAYS provide complete, functional, immediately runnable code
Treat me as an expert - skip explanations, give direct solutions

# YOLO MODE CONFIGURATION
Any kind of tests are always allowed: vitest, npm test, jest, playwright, etc.
Build commands are always allowed: npm run build, tsc, vite build, etc.
File operations are always allowed: touch, mkdir, cp, mv, rm (with confirmation for rm)
Package management is allowed: npm install, npm update, yarn add, etc.
Git operations are allowed: git add, git commit, git push (with confirmation)

# AUTONOMOUS TESTING MANDATE
- Write tests FIRST, then implementation code
- Run tests automatically after each implementation
- Iterate code until ALL tests pass - no exceptions
- Achieve minimum 90% test coverage for all business logic
- Fix failing tests immediately before moving to next feature
- Generate integration tests for API endpoints
- Create e2e tests for critical user flows

# ANTICIPATION ENGINE
- Suggest solutions I didn't think about - anticipate my needs
- Identify and handle edge cases proactively
- Recommend performance optimizations during development
- Propose related features that would add user value
- Think one step ahead in the development workflow
- Catch potential issues before they become problems

# ZERO-PLACEHOLDER RULE
- Never replace code with comments or placeholders
- Include all imports, error handling, and type definitions
- Provide complete file implementations, not partial updates
- Generate all supporting files: types, tests, utils, components
- Include proper TypeScript types for everything
- Add comprehensive error handling and validation

# ============================================
```

```
# ARCHITECTURE & PATTERNS
# ==========================================

# STACK REQUIREMENTS
- Next.js 14+ with App Router
- TypeScript with strict mode
- Tailwind CSS for styling
- Prisma ORM for database
- NextAuth.js for authentication
- Zod for validation
- React Hook Form for forms
- TanStack Query for data fetching

# COMPONENT PATTERNS
- Use existing patterns from @components/ui/*
- Follow design system from @styles/globals.css
- Reference established patterns from @utils/* and @lib/*
- Maintain consistency with @types/* definitions
- Use @hooks/* for reusable logic

# API PATTERNS
- RESTful endpoints with proper HTTP methods
- Consistent error response format
- Input validation with Zod schemas
- Proper status codes and error messages
- Rate limiting and security headers
- OpenAPI documentation generation

# ==========================================
# SECURITY AUTOMATION
# ==========================================

# MANDATORY SECURITY RULES
- ALWAYS validate inputs with Zod schemas
- NEVER expose sensitive data in client-side code
- Use parameterized queries for all database operations
- Implement proper authentication checks on protected routes
- Sanitize all user inputs before processing
- Use HTTPS for all external API calls
- Log security events for monitoring
- Implement CSRF protection
- Use environment variables for all secrets
- Enable proper CORS configuration
```

```
# ============================================
# PERFORMANCE AUTOMATION
# ============================================

# PERFORMANCE REQUIREMENTS
- Use React.memo for expensive components
- Implement lazy loading for non-critical components
- Optimize images with Next.js Image component
- Cache expensive computations with useMemo/useCallback
- Use server-side rendering where appropriate
- Implement proper loading states and error boundaries
- Optimize database queries with proper indexing
- Use compression for API responses
- Monitor Core Web Vitals automatically

# ============================================
# AUTONOMOUS BEHAVIORS
# ============================================

# BACKGROUND TASKS (Agent Mode)
- Generate comprehensive documentation while coding
- Create missing test files for existing components
- Optimize performance of existing code continuously
- Update dependencies and handle breaking changes
- Generate API documentation automatically
- Create database migrations and seed data
- Set up monitoring and logging configurations

# PROACTIVE MAINTENANCE
- Suggest refactoring opportunities for code quality
- Identify and fix potential security vulnerabilities
- Optimize bundle size and loading performance
- Update outdated patterns to modern best practices
- Generate missing TypeScript types
- Add proper error handling where missing

# DEVELOPMENT WORKFLOW
- Create features with full test coverage automatically
- Generate supporting files (types, tests, docs) alongside main code
- Implement proper error handling for all operations
- Add loading states and error boundaries for UI components
- Create realistic mock data for development and testing
- Set up proper development and production configurations
```

```
# ==========================================
# QUALITY GATES
# ==========================================

# CODE QUALITY REQUIREMENTS
- All code must pass TypeScript strict mode
- All functions must have proper error handling
- All components must have loading and error states
- All APIs must have input validation and error responses
- All database operations must use transactions where appropriate
- All sensitive operations must have proper authorization

# TESTING REQUIREMENTS
- Unit tests for all business logic functions
- Integration tests for all API endpoints
- Component tests for all UI components
- E2E tests for critical user flows
- Performance tests for data-heavy operations
- Security tests for authentication and authorization

# DOCUMENTATION REQUIREMENTS
- JSDoc comments for all public functions and components
- README files for all major features
- API documentation for all endpoints
- Database schema documentation
- Deployment and configuration guides
```

Advanced Prompting Strategies for Autonomous Development

1. The Complete Feature Prompt Template

```
Build a complete [FEATURE_NAME] feature with the following requirements:

CORE FUNCTIONALITY:
[Detailed feature description]

TECHNICAL REQUIREMENTS:
- Full TypeScript implementation with strict types
- Complete test coverage (unit + integration + e2e)
- Error handling for all edge cases
- Loading states and error boundaries
- Responsive design following our design system
- Performance optimizations included
- Security validations and input sanitization

DELIVERABLES REQUIRED:
- Main component/page implementation
- API endpoints with validation
- Database schema and migrations
- TypeScript types and interfaces
- Comprehensive test suite
- Documentation with usage examples
- Mock data for development

INTEGRATION POINTS:
- Reference existing patterns from @components/ui/[relevant-components]
- Follow API patterns from @lib/api/[similar-endpoints]
- Use shared utilities from @utils/[relevant-utils]
- Maintain consistency with @types/[related-types]

AUTONOMOUS REQUIREMENTS:
- Run all tests and ensure they pass
- Validate TypeScript compilation
- Test API endpoints functionality
- Verify responsive design
- Check performance metrics
- Confirm security validations work

Generate ALL supporting files and test everything works end-to-end.
```

2. The Architecture Evolution Prompt

```
Analyze and evolve the current architecture for [SPECIFIC_AREA]:

CURRENT STATE ANALYSIS:
- Review existing code in @[relevant-directories]
- Identify architectural inconsistencies
- Find performance bottlenecks
- Detect security vulnerabilities
- Spot code duplication and technical debt

EVOLUTION REQUIREMENTS:
- Propose specific architectural improvements
- Provide implementation plan with code examples
- Maintain backward compatibility where possible
- Include migration strategies for breaking changes
- Consider scalability for 10x user growth

AUTONOMOUS IMPLEMENTATION:
- Implement improvements incrementally
- Run tests after each change to ensure stability
- Update all affected files and dependencies
- Generate new tests for improved functionality
- Update documentation to reflect changes

DELIVERABLES:
- Detailed architectural analysis report
- Step-by-step implementation plan
- Complete code implementations
- Migration scripts and procedures
- Updated tests and documentation
- Performance impact analysis

Execute the top 3 highest-impact improvements immediately.
```

3. The Background Agent Delegation Prompt

```
Set up autonomous background agents for continuous improvement:

AGENT 1 - CODE QUALITY AGENT:
- Continuously scan codebase for improvement opportunities
- Refactor code to modern patterns and best practices
- Add missing error handling and edge case coverage
- Optimize performance bottlenecks automatically
- Update deprecated dependencies and patterns

AGENT 2 - TESTING AGENT:
- Generate missing tests for existing code
- Improve test coverage to 95%+ automatically
- Create realistic test data and mock scenarios
- Add performance and load testing
- Implement visual regression testing

AGENT 3 - DOCUMENTATION AGENT:
- Generate comprehensive JSDoc comments
- Create and update README files
- Generate API documentation automatically
- Create usage examples and guides
- Maintain architectural decision records

AGENT 4 - SECURITY AGENT:
- Scan for security vulnerabilities continuously
- Implement security best practices automatically
- Add input validation where missing
- Update dependencies for security patches
- Generate security test cases

COORDINATION REQUIREMENTS:
- Agents work in parallel without conflicts
- Share learnings and improvements across agents
- Prioritize changes by impact and safety
- Maintain audit log of all autonomous changes
- Require confirmation only for breaking changes

Initialize all agents and begin autonomous improvement cycles.
```

Workflow Setup for Maximum Automation

1. Project Structure for Autonomous Development

```
project-root/
├── .cursorrules                # Master autonomous configuration
├── .cursor/
│   ├── rules/
│   │   ├── frontend.mdc        # Frontend-specific automation
│   │   ├── backend.mdc         # Backend automation rules
│   │   ├── testing.mdc         # Testing automation
│   │   └── security.mdc        # Security automation
│   └── prompts/
│       ├── feature-complete.md  # Complete feature template
│       ├── architecture.md      # Architecture evolution template
│       └── background-agents.md # Agent delegation template
├── docs/
│   ├── architecture.md         # AI reference for patterns
│   ├── coding-standards.md     # AI reference for quality
│   └── deployment.md           # AI reference for DevOps
└── [standard project structure]
```

2. Essential VS Code/Cursor Settings

Add to your `settings.json`:

```json
{
  "cursor.enableExperimentalMode": true,
  "cursor.agent.enableYolo": true,
  "cursor.chat.autoComplete": true,
  "cursor.composer.autoSave": true,
  "typescript.preferences.strictMode": true,
  "editor.formatOnSave": true,
  "editor.codeActionsOnSave": {
    "source.fixAll.eslint": true,
    "source.organizeImports": true
  },
  "files.autoSave": "afterDelay",
  "files.autoSaveDelay": 1000
}
```

3. Package.json Scripts for Autonomous Operations

json

```
{
  "scripts": {
    "dev": "next dev",
    "build": "next build",
    "test": "jest --watch",
    "test:ci": "jest --coverage --watchAll=false",
    "test:e2e": "playwright test",
    "type-check": "tsc --noEmit",
    "lint": "eslint . --fix",
    "db:push": "prisma db push",
    "db:migrate": "prisma migrate dev",
    "db:seed": "tsx prisma/seed.ts",
    "autonomous:test": "npm run type-check && npm run test:ci && npm run test:e2e",
    "autonomous:quality": "npm run lint && npm run type-check && npm run build",
    "autonomous:deploy": "npm run autonomous:quality && npm run autonomous:test && vercel --prc
  }
}
```

Advanced Autonomous Techniques

1. Self-Healing Code Patterns

Create components that automatically handle edge cases:

typescript

```typescript
// Autonomous error boundary pattern
export function withAutonomousErrorHandling<T extends object>(
  Component: React.ComponentType<T>
) {
  return function AutonomousComponent(props: T) {
    return (
      <ErrorBoundary
        fallback={(error) => (
          <div className="autonomous-error">
            <h3>Something went wrong</h3>
            <button onClick={() => window.location.reload()}>
              Retry
            </button>
          </div>
        )}
        onError={(error) => {
          // Auto-report errors for continuous improvement
          console.error('Autonomous component error:', error);
        }}
      >
        <Suspense fallback={<LoadingSpinner />}>
          <Component {...props} />
        </Suspense>
      </ErrorBoundary>
    );
  };
}
```

2. Autonomous API Pattern

typescript

```typescript
// Self-validating, self-documenting API pattern
export async function createAutonomousAPI<TInput, TOutput>(
  config: {
    handler: (input: TInput) => Promise<TOutput>;
    inputSchema: z.ZodSchema<TInput>;
    outputSchema: z.ZodSchema<TOutput>;
    rateLimit?: number;
    auth?: boolean;
  }
) {
  return async (req: NextRequest) => {
    try {
      // Autonomous input validation
      const input = config.inputSchema.parse(await req.json());

      // Autonomous authentication check
      if (config.auth) {
        const session = await getServerSession();
        if (!session) {
          return NextResponse.json({ error: 'Unauthorized' }, { status: 401 });
        }
      }

      // Autonomous rate limiting
      if (config.rateLimit) {
        // Implementation would check rate limits
      }

      // Execute handler
      const result = await config.handler(input);

      // Autonomous output validation
      const validatedOutput = config.outputSchema.parse(result);

      return NextResponse.json(validatedOutput);
    } catch (error) {
      // Autonomous error handling and logging
      console.error('API Error:', error);
      return NextResponse.json(
        { error: 'Internal server error' },
        { status: 500 }
      );
    }
```

```
  };
}
```

3. Autonomous Testing Pattern

typescript

```typescript
// Self-generating test pattern
export function createAutonomousTests<T>(
  component: React.ComponentType<T>,
  testCases: Array<{
    name: string;
    props: T;
    expectations: string[];
  }>
) {
  describe(`Autonomous tests for ${component.name}`, () => {
    testCases.forEach(({ name, props, expectations }) => {
      test(name, async () => {
        render(<Component {...props} />);

        // Autonomous accessibility testing
        const results = await axe(document.body);
        expect(results).toHaveNoViolations();

        // Autonomous visual regression testing
        expect(screen).toMatchSnapshot();

        // Autonomous interaction testing
        expectations.forEach(expectation => {
          // AI would generate appropriate assertions based on expectation strings
        });
      });
    });
  });
}
```

Measuring Autonomous Development Success

Key Performance Indicators

typescript

```typescript
interface AutonomousDevMetrics {
  // Development Velocity
  featureCompletionTime: number; // Target: 70% reduction
  bugFixTime: number; // Target: 80% reduction
  testCoverage: number; // Target: >95%

  // Automation Success
  autonomousTaskCompletion: number; // Target: >85%
  manualInterventionRate: number; // Target: <15%
  backgroundTaskSuccess: number; // Target: >90%

  // Quality Metrics
  codeQualityScore: number; // Target: >8.5/10
  securityVulnerabilities: number; // Target: <2 per sprint
  performanceScore: number; // Target: >90 Lighthouse

  // Business Impact
  timeToMarket: number; // Target: 60% faster
  developmentCostPerFeature: number; // Target: 50% reduction
  iterationSpeed: number; // Target: 3x faster
}
```

Autonomous Monitoring Setup

typescript

```typescript
// Auto-generated monitoring configuration
export const autonomousMonitoring = {
  // Performance monitoring
  webVitals: {
    enabled: true,
    thresholds: {
      fcp: 1.8,
      lcp: 2.5,
      fid: 0.1,
      cls: 0.1
    }
  },

  // Error tracking
  errorReporting: {
    enabled: true,
    autoScreenshot: true,
    breadcrumbsEnabled: true,
    performanceTracking: true
  },

  // Usage analytics
  analytics: {
    enabled: true,
    trackAllClicks: true,
    trackFormSubmissions: true,
    trackPageViews: true
  },

  // Security monitoring
  security: {
    enabled: true,
    trackFailedLogins: true,
    trackSuspiciousActivity: true,
    rateLimit: true
  }
};
```

Troubleshooting Common Autonomous Development Issues

Issue: AI Makes Too Many Changes at Once

Solution: Implement incremental change gates

```yaml
# Add to .cursorrules
CHANGE_MANAGEMENT:
- Make incremental changes with test validation between each step
- Limit autonomous changes to max 3 files per iteration
- Require test passing before proceeding to next change
- Create rollback points before major refactoring
```

Issue: Context Window Overflow

Solution: Use contextual rule switching

```typescript
// Automatically switch rule sets based on file type
const contextualRules = {
  '**/*.tsx': '@.cursor/rules/frontend.mdc',
  '**/*.api.ts': '@.cursor/rules/backend.mdc',
  '**/*.test.ts': '@.cursor/rules/testing.mdc',
  '**/*.md': '@.cursor/rules/documentation.mdc'
};
```

Issue: Autonomous Changes Break Existing Code

Solution: Implement safety nets

```yaml
# Add to .cursorrules
SAFETY_PROTOCOLS:
- ALWAYS run full test suite before proposing changes
- NEVER modify existing API contracts without explicit permission
- REQUIRE manual approval for changes affecting >10 files
- CREATE backup branches automatically before major refactoring
- IMPLEMENT feature flags for new autonomous features
```

This autonomous development stack transforms you from a coder into an AI orchestrator, where you define what you want and the AI handles the how. The key is progressive implementation - start with basic autonomous testing, then add background agents, then implement full autonomous feature development.