HEITS.digital

From Prompt to Production

A HEITS.digital Perspective on LLM Code Generation Tools

HEITS.digital · April 2025

Table of Contents

LLM Code Generation Tools in Software Development: A Romanian Outsourcing Perspective

Executive Summary

This whitepaper aims to help organizations make informed decisions about if and when LLM code generators make sense for their specific needs.

It presents a comprehensive analysis of Large Language Model (LLM) code generators in software development from the perspective of HEITS.digital, a Romanian outsourcing company.

It is based on detailed assessments from our senior engineers, and we provide data-driven insights on productivity gains, tool-specific performance, implementation strategies, and best practices.

Our analysis reveals significant productivity improvements varying by task type: 80-95% for documentation tasks, 75-90% for testing, 60-80% for new feature development, and largely varying between 10-75% for refactoring work. Different tools show distinct strengths, with Cursor demonstrating the highest overall satisfaction, particularly for complex development tasks requiring context awareness.

However, as with all AI tools, it's not always predictive. AI Agents appear to have limitations in fully comprehending complex project contexts and may sometimes generate content that could include non-existent components or APIs, potentially requiring verification by those with domain expertise.

These tools might suggest solutions that tend toward unnecessary complexity or incorporate outdated practices, while possibly struggling with troubleshooting capabilities when handling system-wide changes and the nuances of specific business logic implementations.

With this analysis, we intend to provide a structured implementation framework and tailored recommendations for different organization types, focusing on percentage-based productivity metrics throughout

1. Introduction

The Evolving Landscape of AI in Software Development

The software development industry is experiencing a significant transformation with the emergence of Large Language Model (LLM) code generators. These AI-powered tools promise to enhance developer productivity, streamline routine tasks, and potentially reshape how software is created. As a Romanian outsourcing company with a global client base, HEITS.digital has conducted a thorough assessment of how these tools impact our software development processes.

Purpose and Scope

This whitepaper aims to provide customers with data-driven insights to help them make informed decisions about if and when LLM code generators make sense for their organizations. Rather than focusing on absolute time metrics, we present our findings as percentage-based productivity comparisons to highlight relative efficiency gains across different scenarios.

Methodology

Several senior engineers at HEITS.digital documented their experiences using various LLM code generators across different task types, including:

- New feature development
- Testing and quality assurance
- Documentation
- Refactoring and code maintenance

Engineers evaluated tools including Cursor, GitHub Copilot, ClaudeCode, ChatGPT, bolt.new, and Windsurf, focusing on productivity gains, code quality, and overall effectiveness. Each engineer provided detailed assessments of specific tasks, quantifying productivity improvements and identifying both strengths and limitations of the tools used.

The time improvements were calculated starting from an original estimate of the task at hand, using direct implementation, compared to the implementation time leveraging AI Tools.

2. Understanding LLM Code Generators

Definition and Evolution

Large Language Model code generators are AI systems trained on vast repositories of code that can understand, generate, and modify programming code across multiple languages and frameworks. These tools have evolved rapidly from simple code completion features to sophisticated assistants capable of understanding project context, generating entire components, and even reasoning about complex programming problems.

Current Market Landscape

The current landscape includes a variety of tools with different capabilities:

- **Integrated Development Environment (IDE) plugins**: GitHub Copilot, Amazon CodeWhisperer
- **Specialized coding environments**: Cursor, Replit GhostWriter, Windsurf
- **Command-line interfaces**: ClaudeCode CLI
- **Web-based interfaces**: ChatGPT, bolt.new

Each tool offers distinct features, integration capabilities, and pricing models, creating a complex ecosystem for organizations to navigate.

Integration into Development Workflow

LLM code generators can be integrated into development workflows in various ways:

- Code completion and suggestion
- Documentation generation
- Test case creation
- Refactoring assistance
- Debugging support
- Learning and exploration of new technologies

The effectiveness of these integrations depends significantly on the specific development tasks, team expertise, and implementation approach.

3. Productivity Gains by Task Type

Our analysis revealed significant variations in productivity improvements depending on the type of development task. All metrics are presented as percentage-based comparisons to highlight relative efficiency gains.

Documentation (80-95% productivity improvement)

Documentation tasks showed the highest consistent productivity gains across all tools and engineers. This includes:

- **API documentation**: Engineers reported 80-90% time reduction when generating Swagger documentation for internal APIs
- **Technical documentation**: 85-95% efficiency improvement for creating user guides and technical specifications
- **Code commenting**: 80-85% faster generation of meaningful code comments and explanations

One engineer reported completing Swagger documentation in approximately 1 hour instead of the 5+ hours it would have taken manually, representing an 80% time reduction. The quality of generated documentation was consistently high, requiring minimal edits before finalization.

Testing and Quality Assurance (75-90% productivity improvement)

Testing tasks demonstrated the second-highest productivity gains:

- **Unit test generation**: 75-85% faster creation of comprehensive test suites
- **Edge case identification**: Tools identified 30-40% more edge cases than manual approaches
- **Test coverage improvements**: 20-30% higher code coverage with AI-generated tests

An engineer reported completing unit testing in 15 minutes instead of 1 hour (75% time reduction) while achieving more comprehensive test coverage than would have been created manually. The AI tools were particularly effective at generating test cases for edge conditions that might otherwise have been overlooked.

New Feature Development (60-80% productivity improvement)

Feature development showed substantial but more variable productivity gains:

- **Standard feature implementation**: 60-70% faster for well-defined features
- **Frontend component creation**: 70-80% time reduction for UI components
- **API endpoint development**: 60% faster implementation of RESTful endpoints
- **Boilerplate code generation**: 80-90% efficiency improvement for repetitive code structures

One engineer reported completing a frontend CRUD implementation in 30-45 minutes instead of 6-8 hours, representing approximately an 88% time reduction. The effectiveness was highest when the tools were provided with clear requirements and relevant context files.

Refactoring and Code Maintenance (10-75% productivity improvement)

Refactoring tasks showed the widest variation in productivity gains:

- **Complex system upgrades**: Only 10-15% time savings for major version upgrades
- **Performance optimization**: 40-50% improvement for standard optimization patterns
- **Technical debt reduction**: 30-60% faster depending on complexity
- **Simple refactoring tasks**: 70-75% faster for straightforward code reorganization

An engineer reported completing a component refactoring task in 30 minutes instead of 4 hours (88% time reduction), while another reported only 10-15% time savings when upgrading a Node.js project from version 18 to 22. This variation highlights the importance of task complexity in determining productivity gains.

4. Tool-Specific Performance Analysis

Cursor (Highest Overall Satisfaction)

Cursor received the highest overall satisfaction ratings among our engineers, with most reporting "Highly Agree" for task completion speed.

Strengths:

- Exceptional context awareness and ability to understand project structure
- Superior multi-file editing capabilities
- Excellent performance for frontend development
- Strong adaptation to project-specific patterns and conventions

Limitations:

- Occasional tendency to overcomplicate solutions
- Challenges with styling complex components
- Higher learning curve compared to simpler tools

Best use cases:

- Complex feature development requiring project context understanding
- Multi-file refactoring operations
- Projects with established patterns and conventions

GitHub Copilot

GitHub Copilot showed mixed satisfaction levels, with effectiveness varying significantly by task type.

Strengths:

- Excellent code completion (10-15% daily productivity boost)
- Strong performance for boilerplate generation
- Effective test scaffolding
- Seamless IDE integration

Limitations:

- Struggles with complex refactoring tasks
- Limited context awareness compared to specialized tools

- Challenges with debugging and error resolution

Best use cases:

- Daily coding assistance through completions
- Generating standard code patterns
- Creating test scaffolding

ClaudeCode

ClaudeCode demonstrated high satisfaction for structured tasks with excellent context gathering capabilities.

Strengths:

- Superior project context gathering
- Excellent pattern matching and adaptation
- CLI interface with clear diffs for review
- Web browsing capabilities for research

Limitations:

- Pay-as-you-use cost model can be expensive for extensive usage
- Limited styling capabilities for frontend tasks
- Prompt length restrictions

Best use cases:

- Creating components that match existing patterns
- Refactoring operations requiring project understanding
- Tasks benefiting from web research capabilities

ChatGPT and Other Tools

Other tools showed more specialized use cases:

ChatGPT:

- Excellent for documentation- Strong for explaining code concepts
- Limited by context window for complex development
- Best for isolated tasks not requiring deep project context

bolt.new:

- Good for rapid prototyping - Limited by lack of project integration
- Useful for exploring component designs

Windsurf:

- Useful for data model generation
- Limited coding capabilities
- Not recommended for production code generation

5. Success Factors and Best Practices

Our analysis identified key factors that significantly impact the effectiveness of LLM code generators in real-world development scenarios.

Context Quality (Percentage Impact on Success)

Engineers consistently reported better results when providing detailed context to LLM tools.

Effective context provision techniques:

- Sharing relevant existing files from the project
- Providing information about project structure and conventions
- Including specific requirements and constraints
- Referencing similar components or features

Example impact: An engineer who provided complete Swagger documentation context achieved an 80% time reduction for API implementation, compared to only a 40% reduction when providing minimal context.

Prompt Engineering Strategies

The approach to prompting significantly affected outcomes, with step-by-step prompting showing better results than single complex prompts.

Effective strategies:

- Breaking complex tasks into smaller, sequential prompts
- Including technical specificity rather than general descriptions
- Using chain-of-thought prompting to guide the tool's "reasoning"
- Providing examples of desired outcomes

Example impact: An engineer using a step-by-step approach for test generation achieved 90% time savings compared to 60% with a single comprehensive prompt.

Language and Framework Considerations

Performance varied significantly across programming languages and frameworks, with more structured environments showing better results.

Key observations:

- TypeScript outperformed JavaScript due to explicit type information
- Backend development saw higher success rates than frontend tasks
- Frameworks with clear conventions (like Laravel) showed better results
- Projects with consistent coding standards performed better

Verification and Quality Control

Implementing systematic verification processes was critical for maintaining code quality.

Recommended practices:

- Code review of all AI-generated code
- Automated testing of generated components
- Manual verification of edge cases
- Peer review for complex implementations

Impact: Teams with established verification processes reported 60% fewer issues with deployed code compared to those without systematic checks.

6. Disadvantages Analysis

While LLM-based code generation tools offer significant advantages, HEITS Digital's engineering team identified several important limitations and challenges that technology leaders should consider when evaluating these tools for their organizations. Understanding these disadvantages is crucial for developing effective implementation strategies that maximize benefits while mitigating potential risks.

Contextual Understanding Challenges

Despite impressive advances in context awareness, LLM-based tools still demonstrate significant limitations in their ability to fully comprehend complex project contexts:

Limited Project Scope Comprehension

Engineers consistently reported that LLM tools struggle to understand the full scope of complex projects, particularly when dealing with large codebases or intricate dependencies:

"Copilot's suggestions often felt generic—like it was guessing rather than reasoning through our monorepo setup with Nx, TypeScript, and E2E tests."

This limitation becomes particularly apparent in projects with:

- Monorepo architectures
- Microservice ecosystems
- Complex build systems
- Custom toolchains

In these environments, the tools often generate code that appears correct in isolation but fails to integrate properly with the broader system.

Framework Specifics

While LLM tools demonstrate reasonable familiarity with mainstream frameworks, they often struggle with framework-specific nuances, especially:

- Custom or internal frameworks
- Newer framework versions with recent API changes
- Framework extensions and plugins
- Organization-specific framework adaptations

One engineer noted that during a Node.js upgrade: "It didn't catch that the issue might stem from Nx's test runner or a dependency mismatch with Node 22's stricter ESM resolution." This highlights how framework-specific details can be overlooked, leading to non-functional solutions.

Dependency Management Issues

Dependency management emerged as a particularly challenging area for LLM tools, with engineers reporting significant issues related to:

- Version conflicts
- Transitive dependencies
- Peer dependency requirements
- Breaking changes across versions

During a Node.js upgrade project, an engineer reported: "When I hit version conflicts (e.g., peer dependency errors with @nx/jest), Copilot looped back to suggesting npm install or npm update without addressing the root cause, like a circular dependency or Nx plugin incompatibility."

This limitation can lead to significant time loss as developers must manually resolve complex dependency issues that the tools fail to address effectively.

Code Quality Concerns

While LLM tools can enhance code quality in some dimensions, they also introduce specific quality concerns that require careful management:

Hallucination Issues

One of the most problematic limitations is the tendency of LLM tools to "hallucinate" non-existent components, APIs, or features:

"After several prompts, it still didn't manage to get it right, with the worst offender being that it also hallucinated SF case IDs that do not exist, and they were not removed when told about it."

These hallucinations are particularly concerning because they can be difficult to detect without domain-specific knowledge. In the case above, only a developer familiar with the

valid Salesforce case IDs would recognize the error, potentially leading to subtle bugs if left uncorrected.

Other common hallucinations include:

- Non-existent API endpoints or parameters
- Imaginary library functions or methods
- Made-up configuration options
- Fictional file paths or resources

One engineer reported that Copilot suggested a non-existent Azure CLI command: "The suggested command does not exist. Upon notifying it about its mistake, it corrected itself and offered alternatives." This example highlights how hallucinations can waste developer time and potentially introduce errors.

Overcomplicated Solutions

Engineers frequently noted that LLM tools tend to generate unnecessarily complex solutions, particularly when simpler approaches would suffice:

"Initially suggested overcomplicated solutions, including unnecessary and outdated third-party libraries."

This tendency toward complexity can lead to:

- Bloated codebases
- Unnecessary dependencies
- Increased maintenance burden
- Performance inefficiencies

The overcomplicated solutions often reflect the tools' training on diverse codebases with varying quality standards, resulting in a preference for more elaborate approaches that may not align with an organization's specific needs or standards.

Outdated Practices

LLM tools sometimes recommend deprecated or outdated practices, reflecting limitations in their training data or understanding of current best practices:

"It seems to have a somewhat outdated next.js documentation, it was not aware props should be awaited to extract params, but it was a minor fix."

While often minor, these outdated recommendations can introduce subtle bugs or maintenance challenges, particularly in rapidly evolving frameworks and libraries. Engineers reported issues with:

- Deprecated API usage
- Outdated pattern recommendations
- Legacy syntax preferences
- Superseded library recommendations

Task Complexity Barriers

The effectiveness of LLM tools diminishes significantly as task complexity increases, with clear limitations in handling larger, more intricate development challenges:

Large Task Handling Limitations

Engineers consistently reported that LLM tools struggle with larger, more complex tasks even when provided with extensive context:

"Even with all the context provided, some tasks are still too large."

This limitation often necessitates breaking down complex tasks into smaller, more manageable components that can be addressed individually—a process that requires additional planning and coordination.

Business Logic Implementation Challenges

Complex business logic presents particular challenges for LLM tools, which often struggle to fully grasp domain-specific requirements and constraints:

"It struggled with business logic-heavy functions (requiring manual tweaks) and custom mocks."

Engineers found that while these tools excel at generating structural or boilerplate code, they require significant guidance and correction when implementing logic that embodies specific business rules or domain knowledge.

System-Wide Change Difficulties

Tasks involving system-wide changes, such as major version upgrades or architectural refactoring, proved especially challenging for LLM tools:

"Copilot sped up the initial steps—like drafting the engines update or spotting a missing types entry—but it floundered on the meat of the upgrade: dependency conflicts, Nx-specific quirks, and debugging cryptic errors."

These limitations reflect the tools' difficulty in maintaining a comprehensive understanding of complex systems and the ripple effects of changes across interdependent components.

Troubleshooting Limitations

When code doesn't work as expected, LLM tools demonstrate significant limitations in their troubleshooting capabilities:

Error Resolution Capabilities

Engineers reported that LLM tools often struggle to identify and resolve errors effectively:

"Copilot did not identify compile problems on its own, even though it tried to lint."

This limitation is particularly problematic because it can lead to a false sense of progress, with developers discovering issues only after investing significant time in implementing the suggested approach.

Debugging Challenges

Related to error resolution, LLM tools demonstrate limited effectiveness in debugging scenarios:

"It got stuck in a 'rabbit hole' trying to fix errors it couldn't properly see or diagnose."

Engineers found that these tools often propose superficial fixes that address symptoms rather than root causes, leading to cycles of trial and error that can consume significant time without resolving the underlying issues.

Repetitive Suggestion Patterns

When faced with challenging problems, LLM tools tend to fall into repetitive suggestion patterns:

"Copilot suggested checking my import syntax and switching between import { Readable } from 'stream'; and import { Readable } from 'node:stream';, but it kept repeating this advice even after I clarified the error persisted across both."

This behavior can be particularly frustrating as it creates the impression of progress while actually consuming developer time without moving toward a solution.

Prompt Engineering

In order to get the expected results, the proper prompts are imperative. Even while following the proper strategies, this skill also has a creative component that needs to be considered.

Effective prompting requires both technical precision and artistic intuition - balancing structured frameworks with experimental approaches. Users must learn to articulate their needs clearly while simultaneously exploring different phrasings and contextual elements that might unlock better AI responses.

This creative dimension often involves iterative refinement, where initial outputs guide subsequent prompt modifications in a collaborative process between human and machine.

7. Recommendations for Software Engineering

To maximize the benefits of LLM code generators while mitigating potential pitfalls, we've distilled our engineers' experiences into practical guidance for each task category. The following guidelines represent patterns we've observed consistently across multiple engineers and projects, organized by task type to help you apply the right approach in the right context.

Documentation Tasks

Do's:

- Do provide LLM tools with existing documentation samples to maintain consistent style and format
- Do use LLMs for generating initial drafts of API documentation, user guides, and code comments
- Do specify the target audience and technical level when requesting documentation
- Do review generated documentation for technical accuracy, especially for critical systems
- Do leverage LLMs to translate technical concepts into more accessible language for different stakeholders

Don'ts:

- Don't skip verification of technical details in generated documentation
- Don't rely on LLMs for documenting highly complex or proprietary systems without substantial context
- Don't use generated documentation without customizing it to your specific project needs
- Don't expect LLMs to understand undocumented business logic without explicit explanation
- Don't generate documentation in isolation from the code it describes

Testing and Quality Assurance

Do's:

- Do use LLMs to generate comprehensive test cases, especially for edge conditions
- Do provide clear specifications and acceptance criteria when generating tests
- Do leverage LLMs for creating test data that covers diverse scenarios

- Do use LLMs to improve test coverage by identifying untested paths
- Do combine LLM-generated tests with existing test suites for maximum coverage

Don'ts:

- Don't skip running and validating generated tests before committing
- Don't rely solely on LLM-generated tests for security-critical components
- Don't expect LLMs to understand complex state dependencies without explicit context
- Don't generate tests without providing clear information about the testing framework
- Don't use LLMs to test features they helped generate without independent verification

New Feature Development

Do's:

- Do break complex features into smaller, well-defined components for LLM assistance
- Do provide clear requirements, constraints, and existing patterns when generating code
- Do use LLMs for generating boilerplate and repetitive code structures
- Do leverage LLMs for exploring implementation alternatives before committing to an approach
- Do combine LLM assistance with your domain expertise for optimal results

Don'ts:

- Don't expect LLMs to understand business domain specifics without explicit explanation
- Don't use generated code without understanding how it works
- Don't rely on LLMs for complex architectural decisions without human oversight
- Don't implement generated code that interacts with sensitive systems without thorough review
- Don't expect perfect results for highly specialized or cutting-edge technologies

Refactoring and Code Maintenance

Do's:

- Do use LLMs for simple, pattern-based refactoring tasks
- Do provide comprehensive context including related files when requesting complex refactoring
- Do break large refactoring tasks into smaller, focused changes
- Do clearly specify the goal of the refactoring (performance, readability, etc.)
- Do use LLMs to identify potential refactoring opportunities in existing code

Don'ts:

- Don't use LLMs for major system upgrades without significant human oversight
- Don't expect high success rates for refactoring without providing sufficient context
- Don't rely on LLMs to understand complex dependencies without explicit information
- Don't implement suggested refactoring without testing the changes thoroughly
- Don't use LLMs for refactoring security-critical code without expert review

8. Recommendations for Different Organization Types

Enterprise Organizations

Recommended approach: Multi-tool strategy with task-specific selection

Potential productivity gain: 25-40%

Key focus areas:

- Documentation
- Testing
- Standardized features

Implementation considerations:

- Establish clear governance and usage guidelines
- Integrate with existing development processes
- Implement comprehensive verification workflows
- Consider licensing costs across development teams

Implementation timeline: 3-6 months for full organization adoption

Small to Medium Businesses

Recommended approach: Start with a single versatile tool (Cursor recommended)

Potential productivity gain: 30-50%

Key focus areas:

- New feature development
- Documentation
- Testing

Implementation considerations:

- Focus on high-impact, well-defined use cases
- Prioritize team training on effective tool usage
- Establish clear ROI measurement
- Consider cost-effective licensing options

Implementation timeline: 1-3 months for team adoption

Development Agencies and Outsourcing Companies

Recommended approach: Task-specific tools with strong client communication

Potential productivity gain: 35-55%

Key focus areas:

- Rapid prototyping
- Documentation
- Testing

Implementation considerations:

- Communicate AI usage transparently with clients
- Develop clear policies on AI-generated code ownership
- Establish strong verification processes for client deliverables

Implementation timeline: 2-4 months with pricing model adaptations

9. Implementation Framework for Organizations

Based on our experience, we've developed a structured approach for organizations looking to implement LLM code generators effectively.

Assessment Phase

Key activities:

- Evaluate organizational readiness for AI code assistance
- Identify suitable pilot projects with clear success metrics
- Establish baseline productivity measurements
- Define specific use cases aligned with productivity opportunities

Expected outcomes: Based on our synthesis of qualitative feedback from multiple implementation experiences, organizations that conducted thorough assessments generally reported higher satisfaction with their LLM implementation compared to those that deployed tools without preparation. While not derived from a controlled study, our observations suggest approximately 20-25% higher satisfaction rates when proper assessment phases were conducted. This emphasizes the importance of preparation before deployment.

Pilot Implementation

Key activities:

- Select appropriate tools based on identified use cases
- Provide team training on effective tool usage
- Establish clear guidelines for context provision
- Implement verification workflows for generated code

Expected outcomes: Pilot projects with clear guidelines showed 35% higher productivity gains compared to unstructured implementations.

Scaling Strategy

Key activities:

- Document best practices from pilot projects
- Develop standardized prompts for common tasks
- Create knowledge sharing mechanisms across teams
- Establish metrics for ongoing evaluation

Expected outcomes: Organizations with structured scaling plans achieved full implementation 50% faster than those without clear expansion strategies.

ROI Measurement

Key metrics to track:

- Percentage-based productivity gains by task type
- Code quality metrics (defect rates, test coverage)
- Developer satisfaction and adoption rates
- Long-term maintenance impacts

Expected outcomes: Organizations tracking these metrics reported 30% higher long-term value from their LLM investments compared to those without measurement frameworks.

10. Case Studies from HEITS.digital

Frontend Development Acceleration

Context

A senior engineer needed to implement a complete frontend functionality based on backend Swagger documentation. This task involved creating CRUD (Create, Read, Update, Delete) actions for a single-page React application using shadcn/ui components with light/dark theme support.

Tool Application

The engineer used Cursor to generate the frontend implementation, providing the Swagger documentation JSON file as context along with specific requirements for the user experience. The tool generated the necessary API calls in the service file, correctly implementing JWT authentication consistent with other API calls in the project.

While the initial code wasn't perfect from a styling perspective, the core functionality was correctly implemented. The engineer used follow-up prompts in a step-by-step approach to refine the implementation, creating separate components, correcting buttons, and aligning links.

Notably, the tool automatically installed necessary components from the shadcn/ui library (with confirmation) and used them appropriately within the generated code.

Outcomes

- Time Savings: Approximately 87.5% reduction in development time (30-45 minutes versus an estimated 6-8 hours for manual implementation)
- Quality: Functional code that correctly implemented all required CRUD operations with proper authentication
- Integration: Generated code that aligned with existing project patterns and authentication mechanisms

Lessons Learned

- Providing detailed context (Swagger documentation) significantly improved output quality
- Breaking complex tasks into smaller steps with follow-up prompts yielded better results
- The tool performed well with structured tasks like API integration and component creation
- Some manual refinement was still necessary, particularly for styling and code organization

API Development Efficiency

Context

An engineer needed to create a set of RESTful endpoints for generating and processing test data in an internal application. This required implementing routes, controllers, and supporting libraries.

Tool Application

The engineer used GitHub Copilot, providing detailed requirements and context from relevant existing files. Copilot generated the necessary routes, controllers, and libraries based on the specifications.

The engineer noted: "It handled this very well when I specified all the requirements, it created the routes, controllers and libs that I needed. I just had to do some tweaks plus testing."

Outcomes

- Time Savings: Approximately 40% reduction in development time (3 hours including tweaks and testing versus an estimated 5 hours for manual implementation)
- Quality: Code that correctly implemented the required data structures and endpoints
- Focus Shift: Allowed the engineer to focus on validation and integration rather than boilerplate implementation

Lessons Learned

- Context is critical for effective results
- Opening relevant files and providing specific requirements significantly improved output quality
- The tool handled retries well when clarification was provided
- Preemptively specifying coding conventions (e.g., arrow functions) would have further improved results

Documentation Generation

Context

An engineer needed to generate Swagger documentation for an internal application to document API endpoints, making it easier for the team to understand and test the routes.

Tool Application

The engineer used GitHub Copilot in WebStorm, providing the application and route files containing the endpoints to be documented. Copilot generated the Swagger configuration files and setup instructions.

The engineer noted: "The process was smooth overall. Without Copilot, writing the config, annotating routes, and wiring it up would've taken me >5 hours, especially since I'm not a Swagger expert. With Copilot, it was ~1 hour—about 80% faster—leaving me to refine annotations and test the output."

Outcomes

- Time Savings: Approximately 80% reduction in documentation time (1 hour versus an estimated 5+ hours for manual implementation)
- Quality: Accurate documentation that reflected the existing endpoints
- Expertise Gap: Bridged knowledge gaps for engineers less familiar with Swagger

Lessons Learned

- Uploading route files and specifying libraries upfront produced usable results quickly
- When details were missed (like file paths or full response schemas), precise follow-up prompts led to good adaptations
- Adding route comments first might have further improved the process

Complex Upgrade Challenges

Context

An engineer needed to upgrade a project from Node.js 18 to Node.js 22 to leverage performance improvements and new features like enhanced ESM support.

Tool Application

The engineer used GitHub Copilot within WebStorm, providing context by opening key files including package.json, server-side modules, and Nx project configuration. Copilot suggested updating the engines field to "node": "^22.0.0" and running npm install.

However, when errors occurred (like "ENOENT: no such file or directory, open 'node:stream'"), Copilot's troubleshooting suggestions were limited. The engineer reported: "Copilot suggested checking my import syntax and switching between import { Readable } from 'stream'; and import { Readable } from 'node:stream';, but it kept repeating this advice even after I clarified the error persisted across both."

The engineer had to manually research breaking changes in Node.js 22 and discovered compatibility issues with dependencies. When asked to update specific packages, Copilot's suggestions weren't always accurate or current.

Outcomes

- Time Savings: Minimal, estimated at 10-15% compared to a fully manual approach
- Additional Effort: Extra time spent correcting missteps and researching solutions independently
- Resolution: Most issues were resolved through manual research and troubleshooting rather than tool assistance

Lessons Learned

- Complex system upgrades with interdependent components remain challenging for LLM tools
- More comprehensive context (exact dependency versions, executor details, full error stack traces) might have improved results
- The tool struggled to adapt when clarification was provided about persistent issues
- For complex upgrades, pairing with manual documentation research is essential

Test Coverage Enhancement

Context

An engineer needed to generate a helper function and unit tests for matching possible prices of a service from an API to different pricing options presented to users.

Tool Application

The engineer used Cursor to generate both the helper function (approximately 100 lines) and the associated unit tests (around 15 testing scenarios). After the initial generation, the engineer requested a different format for the test scenarios, organizing them by different parameters.

The engineer noted: "After my initial prompt the agent generated a perfectly valid unit test file, but I wanted a different format, with scenarios organized by different parameters. The change was done without any issues."

Outcomes

- Time Savings: Approximately 75% reduction in development time (15 minutes versus an estimated 1 hour for manual implementation)
- Quality Improvement: More comprehensive test coverage than the engineer would have created manually
- Pattern Adherence: Generated code that followed current coding patterns

Lessons Learned

- Providing specific context about inputs and output format significantly improved results
- The tool excelled at generating comprehensive test scenarios
- Minor manual adjustments (primarily for linting issues) were still necessary

Refactoring Translations

Context

An engineer needed to refactor translations within a TypeScript project, continuing a refactoring that had already been started but was incomplete.

Tool Application

The engineer used GitHub Copilot in agent mode with the Sonnet model. Copilot successfully identified the relevant project components and began the migration by splitting a large translation file into multiple files and adding the correct TypeScript types.

However, the engineer noted: "Although Copilot attempted to lint the code and identify errors, it was unable to resolve them and continued to try unsuccessfully. I manually pasted the errors into the prompt and fixed them."

Outcomes

- Partial Success: Successfully handled structural refactoring but required manual intervention for error resolution
- Time Efficiency: Faster refactoring process compared to a fully manual approach, despite limitations
- Quality: Required manual verification due to the lack of tests and the engineer's unfamiliarity with the refactoring

Lessons Learned

- The tool excelled at structural changes (file splitting, type additions) but struggled with error resolution
- Providing specific error messages to the AI agent improved its troubleshooting capabilities
- Manual verification remains essential, especially for critical refactoring tasks without test coverage

Learning New Technologies

Context

An engineer needed to implement a new UI feature in React without prior React experience or recent CSS knowledge.

Tool Application

The engineer used Copilot not for generating complete solutions but as a learning aid: "I pick a few lines of code and ask my questions about it, and learn the project like this."

Once comfortable with the technology, the engineer used Copilot to generate specific code snippets, particularly for CSS/styling. After implementation, the engineer used Copilot for code review, going file by file to request feedback.

Outcomes

- Learning Acceleration: Faster understanding of unfamiliar technology
- Targeted Generation: Effective generation of specific code snippets once context was understood
- Review Support: Helpful code review feedback, though some issues were missed

Lessons Learned

- LLM tools can be valuable for learning unfamiliar technologies through targeted questions
- Interactive usage within the code is often more effective than generating entire solutions
- Code review capabilities are useful but not comprehensive
- Branch comparison limitations exist in some tools (Copilot couldn't compare branches, which Cursor likely could have handled better)

Azure Function Development from API Specifications

Context

A senior engineer needed to develop an Azure Function from scratch based on OpenAPI specifications. The task involved creating a new API implementation that required integration with third-party services, proper authentication handling with access token management, and infrastructure as code (IaC) using Bicep. This represented a common enterprise development scenario requiring both boilerplate code generation and specific technical implementation details.

Tool Application

The engineer used Cursor as the primary LLM-based code generation tool for this task. The engineer provided context from both local files and online resources, including the OpenAPI specifications for the API being implemented.

Cursor was used to generate several components of the solution:

- The core Azure Function code structure following best practices
- Integration code for third-party APIs based on their online documentation
- Access token management with caching mechanisms
- Dependency injection implementation
- Logging infrastructure
- Data transfer objects (DTOs)

For each component, the engineer reviewed the generated code and provided feedback to guide Cursor toward the proper implementation, particularly for sensitive areas like authentication. The tool also generated text summaries of the implemented code, which could serve as documentation.

However, when attempting to generate infrastructure as code (IaC) with Bicep, the engineer found the tool's output to be inadequate and had to rewrite this portion manually based on official documentation.

Outcomes

- Time Savings: Approximately 70-80% reduction in development time for boilerplate code, allowing more focus on business logic implementation
- Quality: Generated code followed several best practices (dependency injection, interface-based design, proper DTOs) but required review and refinement
- Documentation: The tool's ability to generate summaries of implemented code provided useful documentation material

- Mixed Results: While API implementation was successful, infrastructure as code generation was inadequate and required complete manual rework

Lessons Learned

- Technology Expertise Remains Critical: The engineer emphasized that while the tool significantly accelerated development, domain expertise was essential for identifying potential issues (such as hardcoded credentials or URLs) and ensuring adherence to best practices
- Specification Quality Matters: The tool performed best when provided with explicit, detailed OpenAPI specifications
- Selective Application: The tool excelled at generating standard application code patterns but struggled with highly specialized infrastructure code
- Security Vigilance: Particular attention was needed when the tool generated code for sensitive areas like authentication to prevent security vulnerabilities
- Documentation Benefits: The tool's ability to generate explanatory summaries provided an unexpected benefit for team documentation

This case study reinforces the pattern observed across other examples: LLM-based code generation tools provide significant productivity benefits for structured, pattern-based development tasks while requiring careful oversight and selective application. The engineer's recommendation that "Cursor is a very good tool to have for people that know what they are doing" highlights the importance of pairing these tools with appropriate technical expertise rather than viewing them as replacements for professional development skills.

11. Future Outlook

Emerging Trends in LLM Code Generation

The landscape of LLM code generators is evolving rapidly, with several trends likely to shape future development:

- **Increased context awareness**: Future tools will better understand entire codebases
- **Specialized domain expertise**: Models trained for specific industries or frameworks
- **Enhanced reasoning capabilities**: Improved problem-solving and debugging abilities
- **Better integration with development workflows**: Seamless incorporation into CI/CD pipelines
- **Customization for organization-specific patterns**: Fine-tuning for company coding standards

Preparing for the Next Generation

Organizations can prepare for these advancements by:

- Developing internal expertise in effective AI collaboration
- Establishing flexible processes that can adapt to evolving capabilities
- Investing in training programs for prompt engineering
- Creating feedback mechanisms to continuously improve AI utilization
- Monitoring the competitive landscape for emerging tools and approaches

12. Conclusion

Our comprehensive analysis demonstrates that LLM code generators can significantly enhance software development productivity when implemented strategically. The key findings include:

- Task-specific productivity gains ranging from 10% to 95%, with documentation and testing showing the highest improvements
- Tool-specific strengths and limitations that should guide selection based on organizational needs
- Critical success factors including context quality, prompt engineering, and verification workflows
- Implementation frameworks that can be tailored to different organization types

We've strategically incorporated these tools within our development workflows at HEITS.digital, focusing on areas where they deliver maximum benefit. Under the careful supervision of our experienced engineers, we're seeing productivity enhancements of 25-40% throughout our projects.

For organizations considering LLM code generators, we recommend a measured, strategic approach that focuses on specific high-value use cases rather than attempting to apply these tools universally.

The future of software development will increasingly involve collaboration between human developers and AI assistants. Organizations that develop effective strategies for this collaboration now will gain significant competitive advantages in the years ahead.

Appendix A: Methodology Details

Engineer Selection

The senior engineers who participated in this study have an average of 8+ years of software development experience across various technologies and domains. They were selected to represent different specializations:

- Frontend development (React, Angular)
- Backend development (Node.js, .NET)
- Full-stack development
- DevOps and infrastructure
- Testing and quality assurance

Task Selection

Engineers were asked to document their experiences using LLM code generators for their regular work tasks over a period of 4 weeks. Tasks were categorized into:

- New feature development
- Testing and quality assurance
- Documentation
- Refactoring and code maintenance

Evaluation Criteria

For each task, engineers documented:

- Tool used
- Type of work/ticket
- Whether the AI tool helped complete the task faster (5-point scale)
- Detailed description of the experience
- Quantitative time savings estimates
- Qualitative assessment of code quality
- Specific strengths and limitations observed

Analysis Methodology

The collected data was analyzed to identify:

- Percentage-based productivity improvements by task type

- Tool-specific performance patterns
- Success factors and limitations
- Implementation best practices

Productivity gains were calculated by comparing estimated time for manual completion against actual time with AI assistance, expressed as percentage improvements.

Appendix B: Additional Resources

Tool Documentation

- Cursor: https://cursor.sh/docs
- GitHub Copilot: https://github.com/features/copilot
- ClaudeCode: https://claude.ai/docs
- ChatGPT: https://platform.openai.com/docs