White Paper – GenAI cLabs

Applying GenAI to Code Creation

04.23.2025

The Future of Code

Applying GenAI to Code Creation

04.23.2025

Executive Summary

This whitepaper explores the transformative potential of AI-assisted code generation in enterprise software development. By leveraging Large Language Models (LLMs) and Generative AI technologies, we demonstrate how experienced developers can significantly accelerate application development, focus on higher-value creative work, and produce enterprise-grade code with reduced effort. Our case study details how we successfully built an AI-powered application generator that produces complete, functional codebases from natural language requirements.

1. Background & Overview

The New Frontier of Human-Computer Interaction

AI/ML and LLMs have opened a new frontier in human-computer interaction, with Generative AI (GenAI) enabling us to access, acquire, summarize, and create information faster than ever before. The potential of GenAI capabilities to produce greater efficiency and productivity spans industries and practices, from medical research to application building.

Transforming the Software Development Lifecycle

AI-assisted activities allow experienced professionals to abstract labor-intensive tasks that can be rapidly generated using an LLM, enabling them to focus on training & directing the LLM and reviewing & refining outputs. Developing a software application is a complex process consisting of many layered steps and practices that can significantly benefit from the generative powers of LLMs/GenAI.

Piloting a GenAI Application

At Cadmus, we applied GenAI to build starter UI code frameworks, develop service code scaffolding, assist with code reviews, develop test cases, and more. This whitepaper details our journey in developing an AI-powered application generator that creates complete applications from natural language descriptions.

2. How We Started

Evaluating AI Models

We began by evaluating various AI models including GPT-4o, Claude 3.7 Sonnet, and others to understand their capabilities in generating code. Our assessment focused on:

- Code quality and completeness
- Ability to understand complex requirements
- Security considerations in generated code
- Framework and library knowledge
- Error handling and edge cases

Research & Evaluation Methodology

Our evaluation started with a simple "hello world" application, where we provided identical requirements to different models and compared their outputs based on code quality, completeness, security practices, and adherence to requirements.

Piloting Process

After our initial evaluation, we decided to implement a meta application - an AI-powered app generator that would itself be entirely implemented by AI. This approach allowed us to test the capabilities of LLMs in a real-world scenario while creating a useful tool for future development efforts.

3. Set Up & Configuration

After evaluating multiple AI-assisted development environments, we selected Cursor with Claude as our primary platform for several compelling reasons:

Why We Chose Cursor with Claude

1. **Seamless AI Integration:** Cursor provides native integration with Claude, enabling developers to interact with the AI directly within the coding environment without context switching.
2. **Superior Context Management:** Cursor maintains full project context, allowing Claude to understand the entire codebase rather than just isolated snippets. This was crucial for developing a complex, multi-tier application.
3. **Real-time Collaboration:** The platform enabled our team to collectively interact with Claude while maintaining a shared understanding of the evolving codebase.
4. **Code Execution Environment:** Cursor provides an integrated terminal and execution environment, allowing immediate testing of generated code without switching to external tools.
5. **Version Control Integration:** Seamless GitHub integration simplified the process of committing AI-generated code and tracking changes over time.

Our configuration process involved several steps to optimize the AI-coding experience:

1. **Environment Setup:** We installed Cursor and connected it to our Claude API key with appropriate permissions.
2. **Project Structure Initialization:** We created a baseline project structure with common configuration files (package.json, tsconfig.json, etc.) to provide Claude with appropriate context about our development standards.
3. **Context Window Optimization:** We configured Cursor to maximize the context window available to Claude, ensuring it could maintain awareness of previously generated code components even as the project grew.
4. **Custom Shortcuts:** We established custom keyboard shortcuts for common AI interaction patterns, including code generation, explanation, and refinement requests.
5. **Template Library:** We developed a comprehensive prompt template library that became instrumental to our success. This library included:
   - **Component Templates:** Pre-crafted prompts for generating React components with specific patterns (forms, data tables, navigation, dashboards) that aligned with our design system.
   - **API Endpoint Templates:** Structured prompts for creating RESTful API endpoints with proper validation, error handling, authentication, and documentation.
   - **Test Case Templates:** Templates for generating unit, integration, and end-to-end tests with appropriate mocking strategies and edge case coverage.
   - **Framework-Specific Patterns:** Custom prompts tailored to specific frameworks (React, Express, etc.) that incorporated best practices and avoided common pitfalls.
   - **Security Check Templates:** Prompts designed to have Claude review and identify potential security vulnerabilities in generated code.
   - **Documentation Templates:** Prompts for generating comprehensive documentation at various levels (inline comments, README files, API docs).
6. **Shared Knowledge Base:** We stored these templates in a shared repository, allowing team members to use, refine, and extend them throughout the project. Each template contained placeholders for project-specific information that could be quickly customized before being sent to Claude. We also maintained a version history of prompts, tracking which variations produced the best results for specific types of tasks.
7. **Collaborative Workflow:** We established a workflow where one team member would lead the conversation with Claude while others could observe and suggest modifications to prompts or approaches in real-time.

This configuration provided an integrated development experience where Claude functioned as a collaborative pair programmer rather than an external tool, dramatically reducing the friction typically associated with AI-assisted development. Our Template Library in particular evolved to become a core enterprise asset that encapsulated our growing expertise in effective AI collaboration.

4. Step-By-Step Process

Step 1: Prompt Engineering

Our journey began with prompt engineering - perhaps the most critical element of successful AI-assisted development. Our initial attempts at generating the app fell short of expectations, which taught us valuable lessons about the art and science of crafting effective prompts.

> "Working with an AI coding assistant isn't simply about stating what you want - it's about learning to communicate in a way that maximizes the AI's understanding & capabilities.

The Challenge of Prompt Engineering

When we first approached Claude with general requirements for our application generator, the results were underwhelming. The code was functional but lacked cohesion, security considerations were inconsistent, and many implementation details were left as "To-Do's" or stub functions. This experience revealed that working with an AI coding assistant isn't simply about stating what you want - it's about learning to communicate in a way that maximizes the AI's understanding and capabilities.

We quickly realized that effective prompt engineering is a skill requiring deliberate practice and refinement. Some key challenges we encountered included:

1. **Precision vs. Ambiguity:** Vague requirements led to implementation decisions that didn't align with our vision. For example, our initial prompt asked for "GitHub integration" without specifying the exact nature of the integration, resulting in a simplistic file upload interface rather than the comprehensive repository analysis we envisioned. When we refined our prompt to explicitly request "analysis of repository structure to extract coding patterns, architectural decisions, naming conventions, and security practices for incorporation into the generated code," Claude delivered a much more sophisticated implementation that could parse repository contents, identify recurring patterns, extract best practices from documentation, and actively apply these insights during the code generation process. This specificity transformed a basic GitHub connector into an intelligent knowledge extraction system that significantly improved the quality of generated code.
2. **Balancing Detail and Direction:** Too many specific requirements sometimes constrained Claude's ability to suggest optimal architectural patterns, while too few led to implementations that missed critical functionality.
3. **Managing Context Limitations:** As our requirements grew, we needed strategies to prioritize the most important elements within Claude's context window.

Our Prompt Refinement Process

Through an iterative process of trial and error, we developed a systematic approach to prompt refinement:

1. **Start with Core Functionality:** We began with a minimal viable product description focused on the essential functionality.
2. **Analyze Outputs and Identify Gaps:** After each generation attempt, we conducted thorough code reviews to identify missed requirements, security vulnerabilities, or architectural weaknesses.
3. **Refine with Increasing Specificity:** Each iteration added more detailed requirements, focusing on areas where previous generations fell short.
4. **Incorporate Technical Constraints:** We learned to explicitly state our technology preferences and architectural boundaries.

5. **Include Non-Functional Requirements:** Security, performance, and code quality expectations needed explicit mention to ensure they received proper attention.

This refinement process not only improved the quality of Claude's output but also pushed us to clarify our own thinking about the application. In several instances, the process of articulating requirements precisely led us to discover additional features that would enhance the app's utility.

For example, when specifying how users would access the generated code, we initially only considered local file downloads. The process of detailing this requirement led us to add the option to automatically push generated code to a GitHub repository - a feature we hadn't initially considered but that significantly enhanced the application's value proposition.

Step 2: Development Process

Claude generated the entire application framework, including:

- A React frontend using Tailwind CSS
- A Node.js backend using Express.js
- In-memory storage in lieu of a database (for prototype purposes)
- API integration with AI services
- GitHub repository integration
- User interface for inputting requirements and selecting frameworks
- Code generation pipeline

The development process became a series of iterative refinements, with each generation cycle bringing us closer to a fully functional application.

Interactive Review and Refinement

After each code generation phase, we conducted comprehensive code reviews, examining not just syntax and structure but also architecture, security practices, and completeness. This review process revealed both strengths and limitations in Claude's implementations.

We cloned the generated codebase and followed the README instructions to build and run the application locally, which provided the most reliable indicator of functionality. During these tests, we discovered various issues that weren't apparent in code review alone:

1. **Missing Dependencies:** In early iterations, the package.json file sometimes lacked necessary dependencies that were referenced in the code.
2. **Outdated API Calls:** Claude occasionally used outdated API patterns for libraries.
3. **Incomplete File Structure:** Some important files mentioned in imports were not included in the generated output, requiring us to prompt Claude for the missing components.
4. **Integration Gaps:** The connections between frontend components and backend APIs sometimes contained subtle mismatches in data structures or endpoint URLs.

Each of these issues became the subject of targeted prompt refinements, where we would describe the specific problem and ask Claude to provide corrections or generate the missing components. With each iteration, the application became more complete and functional.

The infrastructure-as-code components followed a similar pattern of improvement. Initially, the Terraform deployment scripts contained optimistic assumptions about cloud environment configurations. When we attempted to apply these scripts in a test environment, we encountered permission issues and missing resource references. By bringing these specific challenges back to Claude with detailed error messages, we gradually refined the infrastructure code until it successfully deployed our application to Azure.

By our final iteration, Claude had generated a complete, deployable application that met all our core requirements. The React frontend successfully communicated with the Express backend, which in turn could process user requirements, integrate with GitHub repositories, extract coding patterns, and generate new application code aligned with the identified best practices.

This iterative dialogue between human review and AI generation proved to be the most effective development methodology. Each cycle improved not just the code itself but our understanding of how to communicate effectively with the AI to achieve our desired outcomes.

5. The Outcomes

What We Achieved So Far

Our AI-assisted development approach yielded several significant benefits:

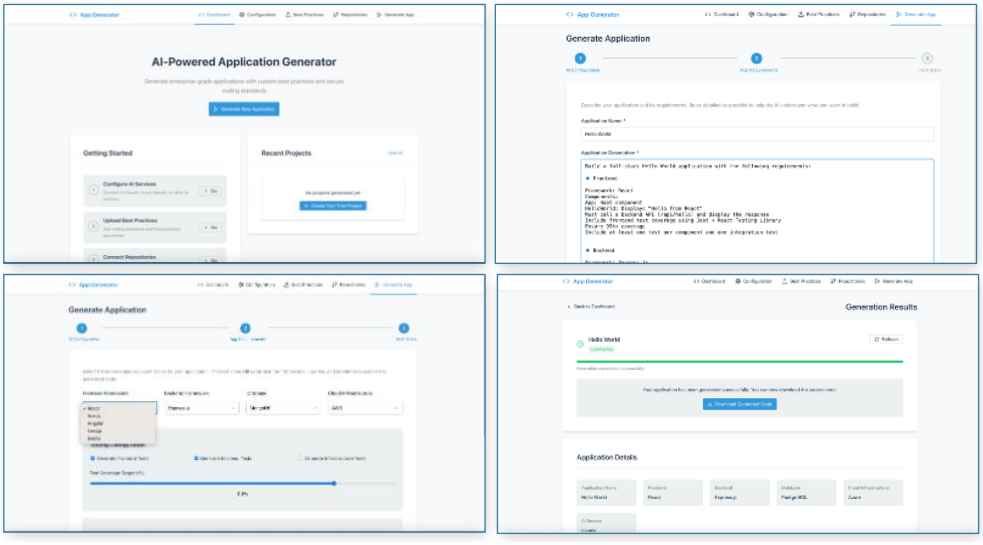
1. **Unlimited Innovation:** The ability to dream big and innovate with only imagination as the limiting factor. In film analogy parlance, developers become "directors" of the code - prompting and wielding AI to bring to fruition the most innovative ideas.
2. **Flexibility and Agility:** The flexibility to adjust and change course as the whole application is developed by the AI agent eliminated traditional constraints around time and cost of rework.
3. **Empowered Development:** This opened an entirely new space for creativity, especially for experienced software engineers who can effectively guide AI agents by asking the right questions and directing them toward optimal solutions.
4. **Accelerated Development Cycle:** We no longer need to progress linearly from POC to MVP to feature expansion. AI enables engineers to think big and execute big from the start.
5. **Enterprise-Grade Code:** The generated code followed security best practices, included appropriate error handling, and maintained a clean architecture.
6. **Dramatic Productivity Gains:** We achieved a 4-6x increase in development velocity compared to traditional coding approaches. What would have taken a team of developers several weeks to build was completed in just days. This acceleration represented a fundamental shift in how quickly complex system designs could move from concept to working implementation, with each iteration cycle compressed from days to hours.

6. Key Takeaways

The Role of the Developer as "AI Director"

We discovered that experienced software engineers are still essential as "directors" of the AI agent. Some examples where human expertise was crucial:

1. **Optimizing Approaches:** Claude initially suggested cloning entire GitHub repositories locally when we wanted to integrate best practices documents stored in repos. After our guidance to use the GitHub API to analyze files, it produced a much more optimized solution.
2. **API Design:** In one instance, the AI suggested a separate API endpoint instead of merging functionality into an existing endpoint, which would have resulted in multiple API calls rather than a single, efficient call.
3. **Library Knowledge:** The AI agent wasn't always up-to-date with breaking API changes in libraries. On several occasions, we encountered runtime errors because external libraries expected different parameters than what the AI had implemented.



Screenshots of the GenAI Application Builder Created using Cursor with Claude

Development Freedom

As "code directors," developers now have more time to think about the best possible innovative solutions without worrying about implementation time and cost constraints. This enables a two-phase approach:

- **Phase 1:** Simple implementation to achieve core functionality
- **Phase 2:** Add advanced features and optimizations

10 Key Considerations for Effective AI-Assisted Development

1. **Precise Prompting:** Be exact in your prompts. For example, when we requested code to pull best practices from "a" GitHub repo while thinking of multiple repos, Claude generated code for a single repo integration. Our Prompt Template Library proved invaluable here, as templates underwent multiple iterations to eliminate ambiguities and edge cases. By maintaining a library of precisely worded prompts that had proven successful, we reduced misinterpretations and ensured consistent outputs across similar tasks. When a prompt produced unexpected results, we refined it and added the improved version to our template collection.
2. **Detailed Instructions:** Don't be afraid to be as detailed as possible in your prompts to ensure the AI understands your exact requirements. Our most successful template prompts often exceeded 500 words, containing specific formatting requirements, error handling expectations, naming conventions, and architectural patterns. The Prompt Template Library allowed us to reuse these detailed instructions without having to recreate them for each interaction. For complex components, our templates included examples of desired outputs alongside explanations of why certain patterns were preferred, giving Claude multiple ways to understand our requirements.
3. **Iterative Refinement:** Don't hesitate to modify instructions based on the results of the code generated by the agent. This is where your experience as a software engineer adds significant value.
4. **Active Participation:** Don't leave all decision-making to the AI agent. Be an active participant in the development process.
5. **Collaborative Problem-Solving:** Suggest better ways to implement functionality or engage in conversation to identify optimal approaches.
6. **Code Quality Oversight:** As software engineers, you need to detect "code smells" and address them in conversation with the agent.
7. **Strategic Token Usage:** Don't use the agent to fix runtime errors that you can easily address yourself. This consumes precious tokens and can be inefficient.
8. **Request Alternative Solutions:** For critical functionality, always ask the agent for alternative implementation approaches, then decide what makes the most sense for your specific context.
9. **Division of Labor:** Use the AI agent for heavy lifting of writing mundane code while maintaining control over the creative and architectural aspects of the solution.
10. **Collaborative Mindset:** The best outcomes emerge when development becomes a collaborative session between human and AI. Every session should feel like wielding a "brush" to create a "beautiful canvas" - a sense of accomplishment where you feel you created the app with AI assistance.

7. Future Directions

Expanding AI-Assisted Development Capabilities

Building on our success, we plan to enhance our AI-powered app generator with:

- Support for additional frameworks and cloud platforms
- More sophisticated code optimization
- Integration with CI/CD pipelines
- Enhanced security scanning and compliance verification
- Custom component and pattern libraries

Enterprise Adoption Strategy

To facilitate wider adoption within enterprise environments, we're developing:

- Training programs for developers to become effective "AI directors"
- Governance frameworks for AI-generated code
- Integration with existing development toolchains
- Metrics for measuring productivity gains

8. Conclusion

AI-assisted code generation represents a paradigm shift in software development. Rather than replacing developers, it elevates their role to that of creative directors who can realize their visions more rapidly and comprehensively than ever before. The most successful implementations will be those that recognize the symbiotic relationship between human creativity and AI capabilities.

By following the best practices outlined in this whitepaper, development teams can harness the power of AI to dramatically accelerate development cycles, reduce costs, and produce higher-quality code while freeing human developers to focus on the creative and architectural aspects of software development where they add the most value.

The future of code belongs to those who can effectively collaborate with AI - speaking its language, understanding its strengths and limitations, and directing its capabilities toward realizing human creative vision.