

In today's fast-moving software development world, AI-assisted coding tools that provide code suggestions are increasingly influencing developer workflows. Companies like **PayPal** harness AI-powered coding aids to accelerate development velocity, while organizations such as **H2O.ai** build and integrate AI tools aimed at boosting developer productivity and code quality. But how do product managers and AI leads evaluate these code suggestion features before shipping them widely?

This post unpacks the key considerations and best practices for evaluating an AI feature that writes code suggestions. Drawing on practical frameworks and toolkits — like the **PM Toolkit** calculators and rigorous spreadsheet-based evaluation sheets — we'll explore why the *model itself is not the product*, how to define meaningful evals pre-release, address task-dependent model behaviors, and balance between automation, augmentation, and innovation patterns.

The Model Is Not the Product: Why Context Matters

Every conversation around AI-assisted code completion inevitably narrows to the quality of the underlying model. And that's a mistake. The **AI model is only one component** of a broader developer workflow tool. The actual product is the coding experience shaped by UI, integration with the IDE, error handling, customization [AI PM vs ML engineer](#) options, and importantly — measurable impact on developer outcomes such as *developer speed* and *win-rate* of accepted suggestions.

Before you dive into metrics tied solely to model outputs like accuracy or perplexity, first ask:

- **What do users do today?** Map their current coding and debugging process without the AI feature.
- **How would suggestions fit into this workflow?** Are they augmenting current tasks, fully automating repetitive code writing, or introducing completely new innovation workflows?
- **What user pain points does this feature aim to solve?** Reducing context switching? Speeding up boilerplate writing? Fixing common errors?

By understanding this context, you can craft an evaluation approach focused on product impact rather than model benchmarks alone.

Define an Eval Before Release: Setting Clear Acceptance Criteria

Far too often, AI features ship without defined **acceptance rate** goals or clear failure modes. A robust evaluation starts with setting explicit *ship/no-ship thresholds* around key metrics, aligned to business goals.

Key Metrics to Define and Track

- **Acceptance Rate:** The percentage of AI-generated code suggestions developers accept vs. reject. A critical proxy for suggestion relevance and usability.
- **Developer Speed:** Measurable improvement in time spent coding or completing features with AI assistance vs. baseline.
- **Win-Rate:** Ratio of AI-suggested code snippets that remain in final code without edits or revert.
- **Error Rate / Failure Modes:** Frequency of flawed or buggy suggestions leading to rework.
- **Retention and Usage Metrics:** DAU/MAU ratio, session length, and frequency of suggestions triggered.

Use **PM Toolkit calculators** to help structure your thinking. For example:

- *RICE and ICE scoring* can prioritize which coding domains or suggestion categories to improve first.
- *A/B testing calculators* guide sample sizes for statistically valid feature launches.
- *ROI and business case calculators* translate developer time saved into tangible cost savings or revenue enablement.

These tools coupled with detailed **spreadsheet-based evaluation suites** enable a data-driven approach well beyond anecdotal feedback.

Task-Dependent Model Behavior and Failure Modes

AI models for code completion don't behave uniformly across all coding tasks. They have **task-dependent failure modes** and varying effectiveness:

Task	Expected Model Behavior	Common Failure Modes	Impact on Developer Trust	Boilerplate Code	High accuracy
and speed in generating repetitive code	Stale or non-compliant code, missing project-specific styles	Moderate;	quick wins but may frustrate if style mismatches	Complex Algorithm Completion	Suggested snippets provide helpful scaffolding
Incorrect logic or security vulnerabilities	High;	incorrect suggestions discourage use	API Usage	Suggestions include properly formed API calls	Deprecated function usage or incompatible parameters
High;	impacts correctness and reliability	Debugging Assistance	Recommend fixes or highlight potential bugs	False positives/negatives causing extra effort	Variable; depends on precision of recommendations

Understanding these behavior patterns helps prioritize evaluation efforts, communicate realistic expectations to users, and decide where to invest model retraining or UX improvements.



Automation vs. Augmentation vs. Innovation Patterns

When introducing AI code suggestion features, it's critical to situate them within one of three workflow impact patterns:

1. **Automation:** The AI replaces manual work, such as generating standard class templates or boilerplate code entirely autonomously.
2. **Augmentation:** The AI acts as a smart assistant, providing in-context suggestions developers review and accept, thus speeding up coding while maintaining user control.

3. **Innovation:** The AI enables new ways of coding — for example, generating code from high-level natural language requirements or facilitating cross-language translations.

Each pattern requires different evaluation approaches:

- **Automation** needs rigorous correctness and reliability testing along with kill switches to prevent shipping harmful code.
- **Augmentation** emphasizes measuring acceptance rate and developer speed improvements without compromising user trust.
- **Innovation** demands qualitative research alongside quantitative metrics to assess user adoption and satisfaction.

Companies like **PayPal** often focus on augmentation to accelerate developer velocity safely, while **H2O.ai** pushes more innovation-oriented tooling that reimagines coding experiences.

Leveraging PM Toolkit and Spreadsheets for Rigorous Eval

PM Toolkit provides structured calculators to approach the evaluation comprehensively:

- *TAM/SAM/SOM calculators* assess market opportunity for AI coding features.
- *ROI and business case tools* help justify investment based on improved **developer speed** and decreased cycle time.
- *RICE, ICE, and Kano models* prioritize feature improvements based on impact and effort, factoring in user delight.
- *A/B test and sample size calculators* ensure statistically sound experiments comparing usage with and without the AI feature.
- *NPS and PMF scores* measure developer satisfaction and product-market fit.

In practice, many product leads at companies like **PM Toolkit** advocate building custom spreadsheet eval suites that combine telemetry data (acceptance rates, DAU/MAU), qualitative user feedback, and ROI projections — all refreshed continuously. This approach avoids hand-wavy claims and aligns exec teams on concrete kill-switch thresholds if metrics degrade or risks emerge.

Summary

Evaluating an AI feature that writes code suggestions requires a holistic view beyond raw model metrics. Key takeaways include:

- The model is part of a broader developer product — understand existing workflows first.
- Define acceptance criteria and success metrics like **acceptance rate**, **developer speed**, and **win-rate** upfront.
- Account for task-dependent behaviors and failure modes to prioritize evaluation focus.
- Position the feature within automation, augmentation, or innovation — different patterns demand tailored evals and risk strategies.
- Use proven tools like **PM Toolkit calculators** and custom spreadsheets to drive rigorous, data-backed decisions.

By applying these frameworks, product and AI leaders at enterprises like **PayPal** and **H2O.ai** have successfully navigated complex AI product launches, empowering developers while managing risk. Evaluating thoughtfully

upfront pays huge dividends in shipping quality, trusted code suggestion features that truly enhance developer productivity.

