

AI x Software Development Lifecycle

What's Changing, and What
Startups Can Build

Naman Jain

July 2025



Foreword

Naman Jain, Stellaris Venture Partners

In late 2021, GitHub released Copilot, the first-of-its-kind AI pair programmer. My friends and I integrated it into our workflows almost instantly. The ability to autocomplete code and generate generic functions felt magical. It boosted our output by 10-15%, and it was clear to me even back then: the way we write software would never be the same.

One year later at Stellaris, I had the opportunity to meet over a hundred startups building AI tools for developers. What came as a pleasant revelation was that AI's impact wasn't limited to writing code; it was reshaping every stage of the SDLC (Software Development Lifecycle). With each model release, I saw engineering velocity rise across our portfolio. Aided by tools like Copilot and Cursor, developer productivity surged by over 30% in just a few years. Soon, these once-novel tools quickly became everyday essentials for engineers, just as indispensable as an IDE or version control.

This rapid shift raised a thought-provoking question from my colleague: "What if software development became so easy that a billion people could do it? What happens when there are 1 billion developers instead of 30 million?" That sparked a deep dive into exploring how AI might democratize software creation and automate every part of the SDLC.

What followed was 18 months of research, conversations, and analysis. This report is the product of that exploration. It breaks down how AI is redefining the roles of developers, designers, PMs, testers, and DevOps teams, analyzes which parts of the workflow are most ripe for automation, and maps out where startups can build meaningful, defensible companies. Of course, this report is a point-in-time snapshot. As AI continues to evolve at breakneck speed, some of the challenges, opportunities, and even the ways we evaluate them may shift dramatically.






















Today, the wave of innovation we're seeing in this space is already opening our eyes to just how deep this transformation can go. From code generation to debugging, and design mockups to deployment automation, startups are reimagining the entire software lifecycle, and doing it faster than anyone expected.

On the next page, you'll find a landscape of companies leading this charge, with each one redefining how software gets built, and fueling even greater curiosity about what comes next. Of these, three are part of the Stellaris portfolio: **Kombai**, a frontend execution agent; **Drizz**, a functional testing agent; and **Whatfix**, whose AI Onboarding Copilot streamlines complex SaaS workflows. We continue to remain bullish on the many startups emerging in this space and aspire to be early partners to those shaping the future.





We hope this report challenges assumptions, surfaces opportunities, and evokes the same level of excitement in you as it did for us while writing it.

Landscape













UI/UX

AI Designer Assistant	      
Frontend Execution Agent	      
Zero-Code App Builder	      


















System Design

System Design Thinker	  
System Design Executor	










Code Writing

Specialised Coding Agents	      
Codebase Navigator	    




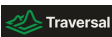











Software Testing

Code Change Impact Analyzers	   
Functional Test Agents	     
Security Testers (Shift-Left Security)	      

Deployment

AI Copilot for Deployment	  
End-to-End Deployment Agent	     

Maintenance

AI SRE	    
AI Onboarding Copilot for complex SaaS workflows	    
Support Ticket Resolution Bot	    

Contents

Chapters are hyperlinked

Context & Evaluation Lens

UI/UX **07**

AI Designer Assistant
Frontend Execution Agent
Zero-Code App Builder

System Design **15**

System Design Thinker
System Design Executor

Code Writing **21**

Specialised Coding Agents
Codebase Navigator

Software Testing **27**

Code Change Impact Analyzers
Functional Test Agents
Security Testers (Shift-Left Security)

Deployment **35**

AI Copilot for Deployment
End-to-End Deployment Agent

Maintenance **41**

AI SRE
AI Onboarding Copilot for complex SaaS workflows
Support Ticket Resolution Bot

Closing Note

Context

For the first time in history, we see a piece of software that can think and write code like a developer. AI models like GPT-4 and CodeWhisperer can now generate functional code snippets, review pull requests, and suggest architectural decisions, tasks that once required technical skill.

Even though we are in the early phase of harnessing this transformative technology, there is immense potential in automating application development due to ever-increasing engineering costs driven by the shortage of high-quality engineers and increasing software complexity.

Startups are uniquely positioned to rethink the software development process, as AI unlocks assistance in areas previously untouched by automation, such as debugging production bugs, generating design inspiration, or proactively testing mobile crashes. While past tools focused on workflow and tracking, AI now enables hands-on execution support and implementation of best practices.

This document breaks down the development lifecycle and explores how AI reshapes the different steps involved in SDLC. It expands on how AI will impact key roles and the startup opportunities arising from these shifts.

What does SDLC entail?

Let's break down SDLC and the stakeholders involved in the process. There are 6 major steps in SDLC:

1. UI/UX Design: Crafting user flows, wireframes, and visual layouts.
2. System Design: Planning architecture, data models, APIs, and scalability.
3. Code Writing: Translating specifications into functional code.
4. Testing: Validating correctness, performance, and user experience.
5. Deployment: Shipping features to production with reliability and safety.
6. Maintenance: Monitoring, debugging, and improving live systems.

And, there are further 5 key personas involved in the development process:

1. Software engineer
2. Designer
3. Product Manager
4. DevOps Engineer
5. Tester

These personas have two archetypes: 'Senior' and 'Junior'.

- The 'Senior' is responsible for managing the team, making decisions on the key elements of the respective steps in SDLC, and solving complex technical problems.
- The 'Junior' is responsible for executing the tasks and resolving implementation issues in the development process.

Evaluation Lens

As part of this report, we identified three core criteria to evaluate the emerging wave of AI developer tools. Our evaluation is based on the type of work the tool supports, its functional scope across use cases, and the type of AI.

Type of Work

Creative Work

Involves finding issues, designing solutions, planning and allocating resources. Demands creativity and critical thinking – qualities that make a task lean toward art.

Execution Work

Involves implementing plans, solving technical challenges, iterating and refining outputs. Requires precision, consistency, and process to bring ideas to life effectively.

Scope

Going Wide

Handles diverse use cases (e.g. GitHub Copilot coding across languages). Trained on huge public datasets, they are ideal for general tasks but less so for specific needs.

Going Narrow

Specialize in focused domains (e.g. SAP Copilot handling SAP workflows). Trained on niche data, they deliver accuracy by limiting scope to defined, repeatable use cases.

Type of AI

Copilot

Assists humans in decision-making (e.g. AI suggesting diagnoses for doctors). Users stay in control via prompts and review. They augment, not replace, human work.

Agent

Independently executes tasks (e.g. AI sales reps managing outreach). Prioritizes action over hints, delivering outcomes in niche domains. Operate with least human input.

In evaluating startup opportunities in AI + SDLC, we also consider:

- Technical feasibility - based on costs, talent, and current research findings
- Ease of GTM - based on competition intensity and barrier to entry
- Market size - based on paying capability and number of customers
- Defensibility - based on data access/storage, depth of the workflow, and habit

In the following six pieces, we will explore how AI is reshaping each step of the SDLC and uncover the opportunities it presents for startups.



1

UI/UX

AI is transforming UI/UX by generating smart design mockups and production-ready frontend code, turning designers into curators and engineers into reviewers. The result is faster workflows, less grunt work, and tighter alignment between design and implementation.

Design workflows today fall into two broad categories, designing from the ground-up, and working within an existing design system, both riddled with their own inefficiencies. Below is a deeper exploration of the challenges in both.

- **Designing from ground up:** When starting something new, whether a product or a new feature, designers face a “cold start” problem. They typically begin by pulling inspiration from visual platforms like Behance, Dribbble, or Pinterest, and manually translate those ideas into Figma or Adobe files. This process is entirely human-driven: ideation, exploration, layout planning, and execution are all dependent on the designer’s experience and available time. There’s no structured system guiding creativity, and there are no tools that actively assist in generating good design starting points.
- **Working within an existing design system:** In mature products, designers operate within a strict visual system, where brand colors, component libraries, spacing rules, and tone are already established. The work is less about creativity and more about extending what's already there: adding a new button to a modal, designing a new state for a card, or combining existing components in a new layout. When components are missing, designers try to follow existing visual patterns to create new ones. This workflow has its own inefficiencies. Searching through a disorganized component library, updating duplicative files, or manually checking brand consistency creates friction. A lot of this is execution overhead, necessary but low-leverage work.

Design-to-Code handoff in the implementation workflow:

Once designs are finalized, front-end engineers inherit the Figma files. But “handoff” is often anything but smooth: engineers must decipher layout intentions, sizing constraints, and interaction logic, often through Slack threads, comments, or meetings. Translating a pixel-perfect layout into production-ready code can take days, mostly due to repetitive boilerplate work like layout scaffolding, theming, responsiveness, and connecting design tokens to code.

What changes with AI

For designers

AI models trained on large corpus of public UI/UX examples (across mobile, web, and desktop apps) can now generate context-aware mockups. Given a feature brief, brand guidelines and platform constraints, an AI system could suggest multiple design options, complete with layout and typography. Instead of starting from a blank canvas, designers would review and customize these AI-generated mockups, either visually or via natural language prompts. Over time, the AI learns organizational preferences, improving both quality and design alignment.

For implementation

Those same AI systems can now generate production-grade frontend code, not just placeholder HTML/CSS, but structured, maintainable React components that follow a company’s design system and use pre-approved code snippets. Frontend engineers shift from writing boilerplate to reviewing, refining, and integrating, making AI a creative collaborator (during ideation) and an execution engine (during implementation), reducing the manual lift required at both ends of the workflow.

Opportunities for startups

1. AI Designer Assistant

Problem

In mature product organizations, design systems are rigid and deeply embedded in how products are built and experienced. Designers are rarely reinventing the wheel. Instead, they're tasked with:

- Scanning the existing component library to see what already exists
- Composing new screens or flows using pre-built components
- Occasionally designing net-new components that conform to brand guidelines
- Collaborating with PMs and engineers to ensure feasibility

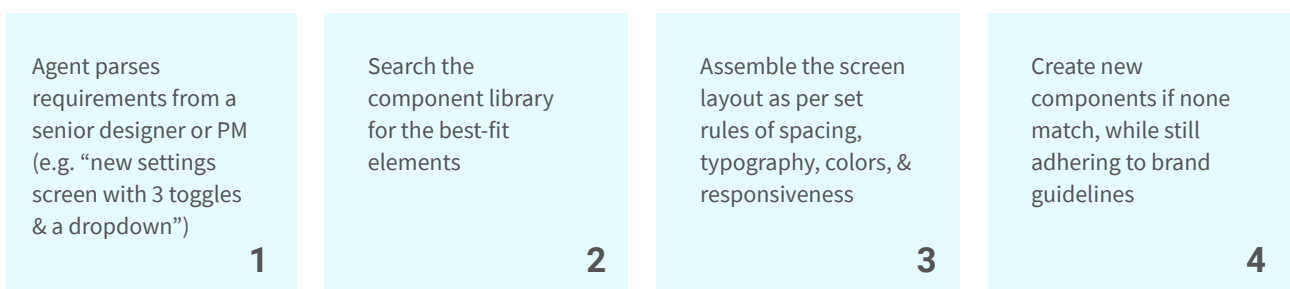
This sounds efficient in theory, but in practice, it's slow, repetitive, and scattered. Component libraries get bloated. Design systems drift. Designers often recreate components that already exist, just styled slightly differently. The system breaks down when it's too big to navigate, and the effort shifts from "designing" to "managing design assets."

Solution

We believe there's a compelling opportunity to build a system that combines two core layers:








- A component library system – a structured, queryable source of truth for all the company's visual components and design patterns.
- An agentic workflow engine – an AI agent that can take in requirements and instantly generate mockups using that component library, fully aligned with the brand's design system.

Think of this AI as a junior designer embedded in your org. It doesn't invent entirely new visual styles, but it excels at execution.



The senior designer's role is evolving toward orchestration, defining goals, setting constraints, reviewing outputs, and providing strategic feedback. Designers and PMs will increasingly rely on general-purpose AI copilots for inspiration, planning, and coordination. The real opportunity for startups lies in building agents to take over the grunt work of visual assembly, executing consistently, instantly, and at scale. This model doesn't replace creativity; it protects it. By offloading execution to AI, designers reclaim time for higher-level design thinking and exploration.

Business Characteristics of AI Designer Assistant

Type of AI	Agentic
Type of work	Creative + Execution Execution - Matching requirements to existing components, assembling UIs Creative - Generating layout variants, and visual explorations within constraints
Scope	Narrow – tied to org-specific design systems
ICP	Senior Designer / PM
Technical feasibility	<p>The hardest part is understanding and aligning with a company’s design system.</p> <p>One solution: Provide a standardized, importable library format, and gradually migrate clients into it. Over time, models can be fine-tuned on company-specific Figma files and internal design tokens.</p>
Ease of GTM	<p>Top-down motion: Target design organizations with 10+ team members, where workflow efficiency matters. Expect longer sales cycles due to integration overhead and data access requirements. Time-to-value may be slow at first, but once embedded, becomes a core part of the design stack.</p>
Market size	<p>Enterprise design teams spend heavily on tools (Figma, Zeroheight, Zeplin, etc.) Cutting a week of design iteration into a day creates a clear ROI.</p> <p>Plausible pricing: \$10–20 per seat/month, with potential for usage-based pricing per project/design</p>
Defensibility	<p>System of record: Anchoring to the design system (i.e., source of truth)</p> <p>Tribal knowledge: Encoded knowledge of how the team designs (e.g., “we always use outlined buttons on mobile”)</p> <p>Habit loop: Designers come back to the agent as it reduces cognitive load & increases creative velocity</p> <p>Integrations: Deep links into Figma, internal docs, PRDs & internal component libraries</p>
Landscape	      

2. Frontend Execution Agent

Problem

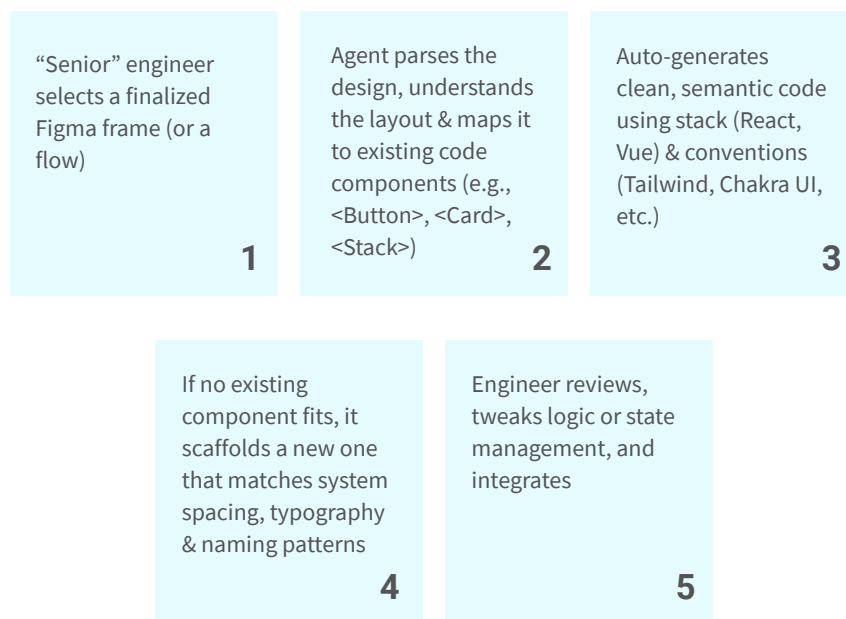
Once a design is finalized, frontend engineers are handed Figma files and expected to turn pixels into production-ready code. This step, while technically routine, is filled with high-friction, low-leverage work:

- Manually recreating layouts with divs, grids, and spacing
- Translating styles into Tailwind or design tokens
- Mapping Figma components to code components
- Ensuring responsiveness, accessibility, and visual fidelity
- Going back and forth with designers to clarify the intent

Even with a solid component library, this is often a tedious grind. Engineers know exactly what needs to be done, but doing it still takes hours. It's not a thinking problem. It's a high-effort translation problem.

Solution

An agentic AI system that acts as a junior front-end engineer who understands your design system, can be transformative. Let's take Kombai (our portfolio) and see how it works:



Kombai's goal isn't to eliminate the engineer, but to remove 80% of the setup work so the engineer can focus on the actual product.

It's like having a trained intern who ships React components that pass code review, uses the right props, and matches your lint rules.

Business Characteristics of Frontend Execution Agent

Type of AI	Agentic
Type of work	Execution: Parsing visual designs, mapping them to code components, and generating structured codebase
Scope	Narrow: Frontend engineers manage a repository that maps code with visual elements for easy asset management. AI needs to be well-versed in code-component pairs along with the best coding and design practices.
ICP	Senior frontend engineer
Technical feasibility	Two core challenges: Disaggregating the hierarchy of objects within Figma, e.g. buttons and text within cards, cards within a page, and figuring out the common elements. Translating Figma components and visual styles to real, usable code. Need a clean and maintained code registry: a mapping between design components and code counterparts. Can start with standardized systems (e.g., Chakra, Tailwind, Material UI) and layer in custom components. Eventually, learn from your git history + component usage to improve accuracy.
Ease of GTM	For Indie Devs & Small Teams: Product-Led Growth Launch as a VS Code extension or Figma plugin Capture dev interest through open-source examples and community demos Let solo devs experience the time savings → land & expand into teams For Enterprises: Top-down motion Target frontend-heavy organizations where component reuse is high and design handoff is a pain point Position as a velocity unlock + consistency enforcer Offer CI integration to catch design drift in PRs
Market size	Frontend is a huge and growing surface area; every product team touches it Engineering orgs already spend \$100s/month per dev on tools (Copilot, Linear, Vercel, etc.) Pricing model: \$10–20/seat/month + optional usage-based pricing (e.g., per screen generated) Core value prop: Cut frontend build times by 50–70%, enforce consistency, and reduce visual bugs, without adding headcount.
Defensibility	Tribal Knowledge: Learns how your team builds UI (naming, layout, theming) Integration Depth: Hooks into your design system, component registry, and repo Workflow Habit: Used daily, embedded into handoff and code generation process Data Moat: Trained on org-specific code and design patterns over time

Landscape



Kombai



Locofy



DhiWise



anima



Codia



builder.io



ion

3. Zero-code App Builder

Problem

Not every application is a VC-scale product. A florist who wants an online booking tool, a retail shop owner trying to run a loyalty program, or an HR manager spinning up an internal dashboard, they all need software but don't have the budget or technical know-how to hire developers or agencies. They're not building for scale; they're building for utility. But today, even a simple app requires navigating confusing low-code tools or stitching together templates.

Solution

This is where an AI-native, end-to-end app builder can change the game.

Imagine telling an AI: "I want a mobile app where customers can book appointments, leave reviews, and get reminders by SMS."

The AI designs the UI, generates the front end, builds the back end workflows, sets up data storage, and deploys the app, all with minimal input.

These apps don't require beautiful design or deep engineering. They need to work, get live fast, and be easy to edit later. The target user doesn't care about clean code or reusable components. They care that their staff and customers can use it without friction.

Today's no-code tools (like Glide, Appgyver, and Adalo) get partway there, but still demand design decisions, data modeling, and logic wiring. AI can collapse all of that into natural language. It can pick the right UI patterns, generate production-grade code behind the scenes, and even host the app automatically.

There are two paths here:

- One is building standalone AI-first tools that let small business owners go from idea to deployed app.
- The other is embedding this capability into vertical SaaS products, offering "app creation" as a feature inside platforms for salons, coaches, logistics operators, etc.

The real unlock is that these users don't need a design system or a repo. They need outcomes. AI becomes the entire builder, not just a copilot, handling design, code, deployment, and hosting in one flow.

Business Characteristics of Zero-code App Builder

Type of AI	Agentic
Type of work	Execution Specifications are defined by the user, AI writes the code and structure everything into one working machine
Scope	Wide: Building apps is a horizontal problem. AI systems that learn to handle layouts, API bindings, and basic logic can generalize across many types of apps. But, initial GTM maybe vertical or use case-specific.
ICP	Non-technical (Business owner and non-IT staff)
Technical feasibility	<p>Large language models are very capable of generating full-stack apps from prompts. Also, most of the underlying capabilities exist individually: UI generation from text, backend code scaffolding, database schema generation, deployment via APIs (e.g., Vercel, Firebase, Supabase).</p> <p>Orchestration is the hard part. Stitching UI, backend, auth, storage, and deployment into a single seamless flow requires significant engineering.</p> <p>Error handling and iteration UX must be airtight; non-technical users won't know how to fix a broken API call or an unexpected layout bug.</p>
Ease of GTM	<p>Going PLG: Non-technical users don't want demos, but need it live in 5 minutes.</p> <p>Huge potential for template-driven growth: "Build an appointment booking app in 60 seconds" or "Create your internal dashboard from a Google Sheet" makes for excellent SEO/social content. Existing low-code/no-code communities are hungry for more power with less complexity; there's an established ecosystem to tap into. Non-technical users are hard to target precisely. Their needs are broad and diffuse. Teams might need to niche down (e.g., "app builder for manufacturers") to break through. This requires strong onboarding, UX, and support, since these users have low trust on no-code as they can't self-debug.</p>
Market size	<p>Enormous long-tail of businesses and teams that need apps but can't hire developers. Millions of small businesses, internal ops teams, and side hustlers. Internal tools market is huge: forms, dashboards, CRMs, inventory trackers, etc.</p> <p>Emerging market for "AI-first operators"—non-devs who want to build fast, iterate, and automate. Willingness to pay is low unless the tool drives direct revenue or saves real hours. Most usage is likely to be broad but shallow—a lot of one-off, simple apps unless you build stickiness into workflows.</p> <p>We predict two types of pricing: App-based (\$50-100 per app) for one-off use cases and monthly subscription for frequent users (\$20-100/mo)</p>
Defensibility	<p>Vendor lock-in: If apps are hosted in the startup's ecosystem, they gain lock-in through data, auth, business logic and updates.</p> <p>Convenient UX: Speed and simplicity are defensible if you get it right. Being 10x easier than existing tools is a powerful moat. Though horizontal platforms are easy to copy unless you have distribution or a sticky UX loop.</p>

Landscape





2

System Design

AI is streamlining system design by recommending patterns, surfacing trade-offs, and auto-generating architecture diagrams and starter code. This reduces bottlenecks and accelerates decision-making without sacrificing quality or governance.

System design is where abstract product requirements are translated into real-world, scalable software architecture. Engineers decide how modules interact, which APIs are exposed, how data is stored and flows through the system, and how security and performance are ensured.

Today, this process is largely human-driven and involves a combination of:

- Whiteboarding potential architectures
- Reviewing existing systems and reusable services
- Figuring out the trade-offs (e.g., monolith vs microservices, SQL vs NoSQL)
- Documenting in tools like Notion, Confluence, or Miro
- Getting reviews from senior engineers or architects before implementation begins

While tools like Lucidchart or Excalidraw help visualize architecture, no intelligence is applied. It relies on tribal knowledge, historical context, and the designer's experience. This creates a bottleneck. Good system designers are scarce, and much of their time is spent on repeatable decisions (e.g., "How should we design a pub-sub system?" or "What's a scalable pattern for CRUD APIs?"). Worse, documentation often lags behind architecture, leading to drift, onboarding friction, and hard-to-maintain systems.

This step is extremely important in SDLC as it lays the foundation for the entire application. If done suboptimally, the application could fail to scale or adapt to future requirements, leading to the wastage of costly developer hours, especially in fast-growing teams where devs do not have time to document properly and analyse four different approaches. With software complexity exploding, there is an alarming need for solutions to aid faster development.

What changes with AI

AI has the potential to ingest large corpora of architectural designs, trade-offs, and best practices across domains to assist with system design. In particular, AI can:

- Recommend system patterns for specific scenarios
- Surface trade-offs between approaches
- Suggest pre-built services and common architectures
- Auto-generate skeletons for system diagrams and starter code

Agentic workflows can also help execute repeatable design tasks like spinning up a service skeleton, setting up basic auth flows, or creating a data model based on a brief, allowing engineers to move faster without missing best practices and governance requirements.

While AI can surface options and generate artifacts, it still struggles with complex trade-offs and long-term system thinking. Early use cases will focus on assisting decisions and automating routine tasks.

Opportunities for startups

1. System Design Thinker

Problem

System design remains a highly manual, experience-driven process. Junior and mid-level engineers struggle to independently own system architecture, relying on senior engineers for mentorship, reviews, and tribal knowledge. This slows down product delivery, reduces engineering velocity, and often leads to inconsistent or suboptimal design choices. Bad early design decisions compound into technical debt and expensive scalability problems later on.

Solution

A reasoning assistant embedded within the engineering workflow that helps engineers think through system design decisions.

Copilot takes product or system needs as input & suggests potential architectures, tools, trade-offs & best practices

1

Instead of automating tasks, it acts as a thinking partner, surfacing alternatives (monolith vs microservices, pub-sub systems)

2

Explains pros & cons, citing documentation or real-world case studies, suggesting the best possible design based on benchmarks

3

Gradually learns from a company's historical design decisions to offer even more context-aware suggestions

4

The copilot helps engineers explore the design space faster and ensures decisions are grounded in known patterns, not just gut feel or tribal knowledge.

Business Characteristics of System Design Thinker

Type of AI	Copilot
Type of work	<p>Creative</p> <p>System design involves critical thinking, evaluation of multiple options, and trade-off analysis. This is fundamentally creative rather than mechanical. Copilot enhances the creative decision-making process, rather than executing predefined tasks.</p>
Scope	<p>Narrow: Best to start with targeted domains (e.g., event-driven architectures, eCommerce applications, video platforms).</p>
ICP	Staff Engineers, Principal Engineers, Distinguished Architects
Technical feasibility	<p>The bar for accuracy in system design is very high, given the critical impact of architectural decisions on the scalability and reliability of business applications. This is one of the hardest technical problems to solve because the best architectural practices and design patterns of leading tech companies are not publicly available for model training. Capturing this opportunity will require innovative strategies to bridge the gap between scant knowledge and actionable recommendations. Success will depend on building strong retrieval mechanisms capable of surfacing architectural patterns, trade-offs, and tool documentation from diverse and scattered sources. This demands sophisticated reasoning models, not just basic RAG pipelines. Complicating the challenge further, architectural knowledge exists in many forms (diagrams, documents, codebases, tribal discussions) making it essential to standardize and map this knowledge into a unified vector space. Finally, for real adoption, the copilot must not merely output diagrams/architectures, but explain reasoning clearly, enabling engineers to trace every step of the design decision-making process.</p>
Ease of GTM	<p>GTM motion will need to be top-down, targeting tech companies with 50–1000 engineers, large enough to have complex systems but still nimble enough to adopt new tooling. Sufficient scale and application sophistication are critical to fully realizing the value of a system design copilot. Historically, system design has not been a formalized area of software development. Engineers have relied on ad hoc methods (whiteboards, documents, blogs, and internal review) to make architectural decisions. As a result, justifying the ROI will be essential, especially by demonstrating improvements in engineering velocity, scalability, and reduction of costly rework. Expect longer sales cycles, driven by integration overheads, access to internal IP, and data privacy requirements for personalization. Deep embedding within existing engineering workflows and strong security assurances will be necessary to overcome buyer hesitation.</p>
Market size	<p>Usage-based: Pay per design project + setup fee for personalized codebase ingestion</p>
Defensibility	<p>Data network effects: As you capture more system designs, decision trees, and trade-offs, you become better.</p> <p>Tribal knowledge: Encoded knowledge of how the team designs (e.g., “we use monolith vs microservices”)</p> <p>Habit loop: Designers are coming back because it reduces cognitive load and increases delivery velocity</p> <p>Integrations: Deep links into Jira, internal docs, PRDs, and internal architecture libraries</p>
Landscape	

2. System Design Executor

Problem

Beyond assisting system reasoning, engineers face a second major bottleneck: the repetitive effort required to translate a high-level design into diagrams, documentation, boilerplate code, and cloud infrastructure templates. Even once a system’s architecture is clear, much time is wasted drawing boxes and arrows, setting up APIs, initializing repositories, or defining infrastructure as code (IaC). This manual work not only slows teams down but also introduces inconsistencies and gaps between design and implementation.

Solution

An agentic solution that follows the below workflow:

Agent automates creation of architecture diagrams, code skeletons & IaC blueprints from textual inputs

1

Engineer describes system needing API gateway & NoSQL database. Agent generates diagram, starter code, Terraform templates

2


Speeds up early stages of building new applications & ensures better consistency within design & implementation.

3

Ultimately, human review is essential to catch edge cases, security misconfigurations & cost inefficiencies.

4

Business Characteristics of System Design Executor

Type of AI	Agentic
Type of work	Execution This work is mostly mechanical once the high-level design is decided. Agents are well-suited for tasks that follow repeatable patterns (drawing components, setting up files).
Scope	Same as System Design Thinker
ICP	Same as System Design Thinker
Technical feasibility	Important to: Ensure the implementation is production-grade and adapts to the style of an organization. Have the right set of evals to limit hallucination and ensure correctness. Build ability to reason and retrieve the right set of data from the training set.
Ease of GTM	Same as System Design Thinker
Market size	Same as System Design Thinker
Defensibility	Same as System Design Thinker
Landscape	

A large, stylized white number '3' is positioned on the left side of the upper half of the page. The background is a dark blue gradient with a complex, abstract pattern of interconnected lines and dots, resembling a network or a molecular structure. The lines are thin and light blue, while the dots are small and also light blue. The overall effect is a high-tech, digital aesthetic.

3

Code Writing

Until recently, coding was slow, manual, and mentally draining, with developers juggling coordination overhead and repetitive grunt work. Now, AI is altering the workflow with code generation, refactoring, documentation, and contextual debugging. Engineers are evolving from code writers to reviewers and system thinkers.

Until recently, software development was a fully manual process, powered by human effort, intuition, and diligence. Developers wrote every line of code by hand, reviewed each other's changes, debugged unpredictable behaviors, and maintained sprawling systems from scratch.

This labor-intensive model made software development expensive, brittle, and fundamentally hard to scale. Codebases grew faster than teams could manage. Reviews were inconsistent, shaped by individual biases. And even high-performing engineering teams found themselves battling avoidable bugs, regressions, and tech debt cycles. Reliability, performance, and maintainability often took a backseat to delivery speed, giving rise to the technical debt.

Individual developers brought valuable context, but were constrained by their own cognitive bandwidth, preferred tools, and prior experience. Collaboration was essential, but also expensive. Large projects often resembled coordination puzzles more than engineering challenges, requiring handoffs between frontend, backend, infra, and QA specialists just to move a feature from spec to prod.

Perhaps most frustrating: the work wasn't always meaningful. By some estimates, 60–70% of a developer's time is spent on what can only be described as digital grunt work, implementing CRUD endpoints, wiring business logic, resolving merge conflicts, fixing lint errors, or scaffolding test cases. It's the kind of work that is essential, but mentally draining, repetitive, error-prone, and rarely creative.

What changes with AI

The rise of foundation models like GPT-4, Claude, and open-weight coding LLMs, marks the first time in history that machines can not only read and write code, but also reason about it. This has shifted AI from a passive tool to an active collaborator.

Modern LLMs, trained on billions of lines of open-source and enterprise code, can now:

- Translate natural language specs into functional code
- Explain and navigate unfamiliar codebases
- Suggest bug fixes and architectural improvements
- Refactor legacy modules into modular, idiomatic patterns
- Auto-generate documentation on the fly

More importantly, they can do all this in context, understanding project structure, following naming conventions, and aligning with team-specific design patterns. In many engineering teams today, code is no longer "written" from scratch, it's assembled, guided, and validated by AI.

This doesn't make developers obsolete. But it redefines their role. Developers are evolving from code generators to reviewers, editors, and orchestrators. Their new responsibilities are:

- Structuring the problem for AI
- Validating correctness and security
- Integrating and refactoring generated code
- Providing feedback loops that make models better over time

Opportunities for startups

1. Specialized Coding Agents

Problem

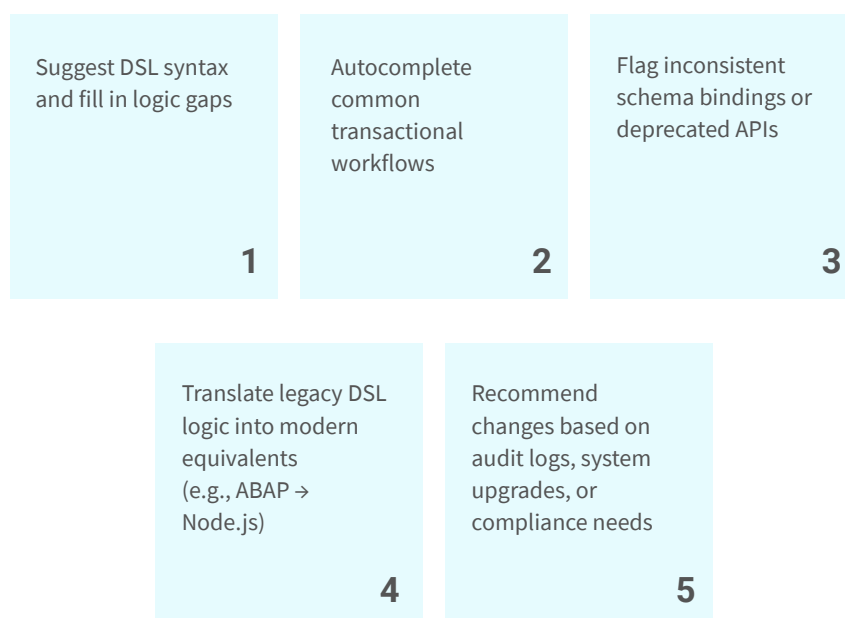
LLMs trained on public repos excel at writing general-purpose code, e.g. Python scripts, React components, Flask APIs, etc. These languages are widely documented, open-source, and have millions of examples in the training data. But this logic breaks down in the world of packaged enterprise applications like SAP, Salesforce, Oracle, or even internal tooling frameworks.

These systems are often black boxes to LLMs. Their scripts are buried in proprietary DSLs, coupled with business logic that's undocumented and unavailable in public training data, making it nearly impossible for general-purpose models to generate useful output.

Even within an organization, only a few engineers understand the implementation deeply. These systems are critical to core operations e.g. billing, CRM, logistics, procurement, but impossible to modernize or extend without expensive consultants or long onboarding cycles.

Solution

A vertical-specific agent, like GitHub Copilot, trained on proprietary code and logic, can assist inside enterprise systems. Once fine-tuned on company-specific scripts, it can:



In practice, this could show up as a VS Code extension for SAP ABAP, or a Salesforce-native agent that understands the internal object schema, workflow automation rules, and system constraints.

Business Characteristics of Specialized Coding Agents

Type of AI	Agentic
Type of work	Execution: The system helps implement and debug well-defined logic in constrained environments, rather than creating novel features or architecture. The core challenge is understanding private DSLs and platform-specific quirks, not creativity.
Scope	Narrow: Every platform (SAP, Salesforce, Oracle, ServiceNow, etc.) has its own syntax, architecture, and ecosystem. Each vertical agent would require deep fine-tuning on internal metadata, object models, and usage patterns.
ICP	Enterprise developers working on packaged platforms (SAP consultants, Salesforce engineers) System integrators modernizing or maintaining enterprise apps Internal tools teams at large orgs running business-critical workflows
Technical feasibility	Medium to Hard: Ingest and index private DSL code, configuration metadata & business logic Build internal code graphs or embeddings for search and traceability Integrate with the enterprise platform's API or plugin architecture (e.g., SAP Business Application Studio, Salesforce DX) Feasible with strong enterprise access, but requires security controls and offline/air-gapped deployments in many orgs. Retrieval-augmented generation (RAG) will be necessary to support context-specific responses at runtime.
Ease of GTM	Hard, but high stickiness once adopted. Pathways to GTM: Sell into System Integrators (TCS, Infosys, etc.): Embed copilots into their modernization toolkit. Difficult due to variable workflows and cultural friction. Full-stack SI Disruption: Build a new-age services firm with AI-native execution. Acquire smaller boutique SIs. Own implementation + tooling. Start Narrow: Offer fast, cheap post-implementation support or projects stalled mid-migration. Deep pain, Lower entry barrier, fast ROI. Challenge: Transitioning enterprise buyers from FTE-based to outcome-based pricing will require cultural and contract innovation.
Market size	Massive: SAP and Salesforce alone support millions of enterprise workflows globally. Just SAP has over 400,000 customers, with many spending millions per year on custom development and consulting. If you can reduce reliance on third-party consultants or cut development cycles by 30–50%, the value delivered per seat is substantial. Verticalized agents can justify premium pricing (e.g., \$100-300/user/month) due to complexity and value delivered. Still early days. No clear category leader has emerged, presenting a rare window for a vertically deep, enterprise-grade agent to own this space.
Defensibility	Vertical tuning: Fine-tuned DSL models per customer or industry (e.g., logistics, pharma, BFSI) are hard to replicate. Speed of implementation: Most of the packaged application coding projects miss deadlines and over-shoot the budget, leading to direct and opportunity cost to the business. Embedding into IDEs and workflows: Integrating directly into SAP Business Studio or Salesforce Flow Builder creates stickiness. Proprietary data moat: The model improves as it ingests internal scripts, metadata, and system logs. Audit & compliance knowledge: Over time, the agents can encode regulatory rules (e.g., SOX, HIPAA) and flag risks proactively, a unique moat in regulated industries.

Landscape       

2. Codebase Navigator

Problem

Imagine a collaborative mural where each artist paints a section of the canvas. If one leaves midway, the replacement must first interpret the brushstrokes, palette, and intent before adding their own. Software development is no different.

When a developer joins a team, or revisits an unfamiliar codebase, they spend hours just trying to understand what exists. Like flossing, documentation is universally acknowledged as important, yet often skipped. Developers prioritize writing and shipping code. Writing clear, comprehensive documentation rarely aligns with sprint goals or KPIs, and often gets postponed indefinitely.

Hence, critical context is often buried in Slack threads, scattered comments, tribal memory, or lost with former team members. This problem intensifies with legacy systems, where outdated tech stacks must be modernized to meet current business needs, but no one remembers how the system was wired. The result is slow onboarding, poor handovers, duplicated logic, and a paralyzing fear of breaking things.

Solution

An AI copilot that acts like a dedicated assistant for developers, automatically generating human-readable and context-aware documentation from the codebase.



Finally, the copilot responds to questions like “Why is this API returning null?”, “Where else is this function used?”, and “What changed in this module since the last release?”

By combining semantic code search with LLM-powered reasoning, it becomes a 24/7 onboarding and debugging companion. Unlike static documentation, it adapts to codebase changes in real time and explains logic with the nuance of a senior engineer. Documentation becomes continuous, always up to date with the codebase, and tailored to your team’s voice and structure.

Business Characteristics of Codebase Navigator

Type of AI	Copilot (with agentic extensions in documentation workflows)
Type of work	Execution: The system performs reasoning and information retrieval based on existing code and context. It does not generate net-new functionality but focuses on summarizing, tracing, and explaining what already exists, hence, it is squarely in the execution domain.
Scope	Narrow to moderate: While the copilot can be deployed across many types of applications, the value increases significantly when tuned for specific systems, frameworks, or domains. For example, understanding a legacy insurance application's claims module requires both code-level parsing and domain-specific heuristics to provide useful insights.
ICP	Engineering managers focused on onboarding and velocity Developer experience (DevEx) teams Teams with high code churn or large, distributed engineering orgs CTOs of growing startups investing in systematizing knowledge
Technical feasibility	Medium: Generating useful summaries from code is technically feasible today, using LLMs and AST (abstract syntax tree) parsing. The harder part is context stitching: tracking changes over time, understanding cross-file dependencies, and aligning with team-specific documentation patterns. Understanding the origin, flow, and endpoints of data within the tangled web of code is critical for providing meaningful documentation. Integrating data sources is key; ingesting unprocessed rough notes, broken docs, Slack comments and vague compliance rules into a model is like putting garbage in a trash can and hoping for it to segregate and recycle on its own. Processing unstructured data from disparate, noisy sources is a non-trivial NLP problem, especially without clear semantics or annotation. Logging user queries and taking feedback from expected output over time will be required to make the application valuable.
Ease of GTM	Medium: Initial adoption may face friction, as the product requires deep access to a company's codebase and associated metadata to deliver real value. Data wrangling and manual curation will likely be needed early on. But once deployed, the product is lightweight to operate, offers fast feedback loops, and integrates naturally with IDEs or internal portals. Suggestive tools are easier to adopt than executional agents, making developer buy-in easier over time. Once embedded in review flows or onboarding checklists, adoption can spread bottoms-up.
Market size	Developer tooling is a \$50B+ market globally. Engineering orgs already pay \$10–20/month for tools like Copilot, Linear, or Sourcegraph. A tool that reduces onboarding time, improves internal DevEx, and improves retention of technical knowledge has clear ROI. Developer onboarding often takes 3–9 months. Cutting that by even 30% can yield savings of \$5,000 or more per engineer. Enterprises with large codebases and high churn stand to benefit most. Over time, this assistant can evolve into a broader developer knowledge platform, enabling code reuse, limiting tribal dependencies & serving as the memory layer. Pricing could be seat-based (\$10–25/user/month) or usage-based (per repo or per doc generated).
Defensibility	Potential to become the “search engine” for your code Tribal knowledge: Tool becomes more valuable as it ingests more of a repo's history. This historical insight enables better explanations. Consequently, deep links into codebase, production env, internal docs, PR repository, and PRDs will be hard to replace. Workflow integration: Becomes sticky by embedding in GitHub, GitLab, or CI flows. Cultural alignment: Learns tone, format, and expectations of the team, becoming a “house style” for documentation. Trust loop: As outputs are corrected, system improves. Feedback cycle becomes tough to clone.

Landscape

  //   

A large, stylized white number '4' is positioned on the left side of the upper half of the page. The background is a dark blue gradient with a complex, abstract pattern of interconnected lines and dots, resembling a network or a molecular structure, in a lighter blue shade.

4

Software Testing

Today's testing is still manual, brittle, and misaligned with developer incentives, leading to gaps, delays, and bugs in prod. AI flips this. By understanding code, PRDs, and edge cases, it can generate, update, and execute tests automatically, making testing faster, smarter, and more complete.

Software testing today consists of two major components: writing and executing test cases. While innovations like record-and-play tools, synthetic data creation, and automation frameworks have improved efficiency, most testing remains manual and error-prone.

Test case generation – whether for backend logic, APIs, or user interfaces – still requires human understanding of the application and its edge cases. Execution, though automated through frameworks like Selenium, Cypress, or Appium, often involves brittle scripts that need continuous maintenance with every product update.

Some companies have implemented automation across devices, browsers, and platforms to improve testing coverage. However, the core challenge in software testing lies in misaligned incentives for developers. Developers are motivated and recognized for building new, high-impact features, not for following testing protocols. As a result, three major bottlenecks persist:

- Coverage Gaps: Many edge cases and negative paths are missed.
- Maintenance Overhead: Test suites must be updated manually with every product or design change.
- Slow Turnaround Time: Test failures often require manual debugging and rework, delaying release cycles.

In short, the current testing process struggles with scalability, reliability, and completeness, leading to costly bugs, security vulnerabilities, and poor user experiences slipping into production.

What changes with AI

LLMs bring a powerful capability to review, generate, and reason, which extends to testing. They can ingest the PRDs, codebase, and existing test suites (whether code or text descriptions) in AI to automatically:

- Generate new test cases covering functional, edge, and negative scenarios.
- Update test suites dynamically as the application evolves.
- Identify gaps in current coverage based on system understanding.
- Execute tests reliably across environments without brittle scripting.

LLMs and agentic workflows open up intelligent error handling: AI systems could diagnose test failures, suggest fixes, or even auto-correct minor issues. This will shorten release cycles, reduce developer dependency, minimize security and compliance risks, and improve customer experience through more robust, error-free applications.

On top of the productivity boost in reaching production-level code, the shift-left movement in QA and DevOps to reduce the high cost of repeated development cycles required to fix production bugs and technical debt will aid the adoption of automated testing tools.

In addition to the startup opportunities identified in this section, testing includes unit testing, which is today addressed by the large, horizontal model players (Github Co-Pilot, Codium, Cursor), and performance testing, solved through frameworks with defined load parameters to test the spectrum.

Opportunities for startups

1. Code Change Impact Analyzers

Problem

Developers struggle to predict how their code changes impact the broader system. Test coverage is often outdated, and new tests are written reactively – after bugs appear – leading to delays and rework.

Solution

An AI co-pilot that has the organizational context, works inside pull requests to predict regressions and suggest tests during the code reviews itself.

Copilot reads the diffs and dependencies from pull requests

1

Highlights risky areas or affected features

2

Suggests relevant unit or integration test cases

3

Keep the test suite updated with each pull request

4

This tool behaves like a senior engineer who knows the whole system and ensures safer, faster merges.

Business Characteristics of Code Change Impact Analyzers

Type of AI	Co-pilot
Type of work	Creative: Co-pilot will take care of diverse scenarios & suggest test cases with 100% coverage.
Scope	Wide: It works across stacks and languages. This tool is responsible for engineering and code-level testing rather than business logic testing.
ICP	Senior Engineer (Tech leads, architects, staff+ level)
Technical feasibility	<p>Medium complexity: Most components (diff readers, static analyzers, LLM co-pilots) already exist; accuracy and context depth are the primary challenges.</p> <p>Impact analysis data: Teaching the LLM to reason & figure out the impact will require examples and workflows across different codebases and applications. Cracking access to historical data, code failures, and the developer workflow to detect and rectify them will be a key win.</p> <p>Static + Semantic Code Analysis: Must understand diffs in the context of entire modules or services, requiring AST (Abstract Syntax Tree) parsing and semantic reasoning across files.</p> <p>Dependency Graphing: To suggest impacted tests or paths, the system must model data flow, control flow, and dependencies across code layers (API, DB, logic).</p> <p>Test Suggestion Generation: Leveraging LLMs for code+test generation is feasible (e.g. GPT-4, CodeWhisperer), but aligning them with custom frameworks (e.g. Jest, RSpec) needs guardrails.</p> <p>Minimal Infra Overhead: Runs in stateless, lightweight environments (e.g., as a GitHub Action or VSCode extension), making it technically easier to deploy and test.</p>
Ease of GTM	<p>Product should fit seamlessly into existing PR workflows, requiring zero context switching.</p> <p>Monetization: Will be complex as you need to win the decision maker (CTO, VP Engg) to get paid and deploy personalized models. Specifically, justifying ROI and pricing the product will be key friction points as the buyers may expect this to be bundled with existing code tools (like Codecov, Snyk, etc.), so pricing must be clear and competitive.</p> <p>Strong PLG potential: One can get started and then shift to an outbound top-down motion post traction. Build early trust with open-source/free tiers and convert to paid via usage-based or team plans. Tools with low integration friction can work as a lightweight GitHub/GitLab plugin or even a CLI tool, making it easy to trial with individual developers or teams.</p> <p>In terms of time-to-value: Instant feedback in the dev workflow = high wow factor. If the AI provides actionable test suggestions or catches regressions, adoption spreads organically. Risk from players in overlapping spaces includes static analysis (\$2B market), linting/code quality tools, and dev productivity platforms (e.g., Sourcegraph, Codacy)</p>
Market size	Modern dev teams have already invested in tools like GitHub Copilot, Sentry, and Codecov. Code intelligence and CI plugins (e.g., CircleCI insights, Code Climate) command \$10–30/seat/month pricing. Broad horizontal applicability across every code-writing team; potential to reach millions of developers globally, implying a \$1–2B+ SAM even before upsells.
Defensibility	<p>Tribal knowledge: The tool becomes more valuable as it ingests more of a repo's history – tracking change patterns, regressions, and test coverage over time. This historical insight enables better impact predictions and smarter test suggestions.</p> <p>Habit loop: Integrating tightly into pull request workflows (GitHub, GitLab, Bitbucket) builds daily user habits. Tools that become part of code review checklists enjoy strong user retention.</p> <p>Trust and Accuracy Loops: If suggestions regularly catch edge-case regressions/coverage gaps, developer trust grows. This feedback loop is hard to recreate without long-term model tuning.</p> <p>Language and Stack Coverage: Expanding support across languages, frameworks, and deployment environments increases surface area defensibility.</p> <p>Integrations: Deep links into codebase, production env, internal docs, PR repository, and PRDs.</p>

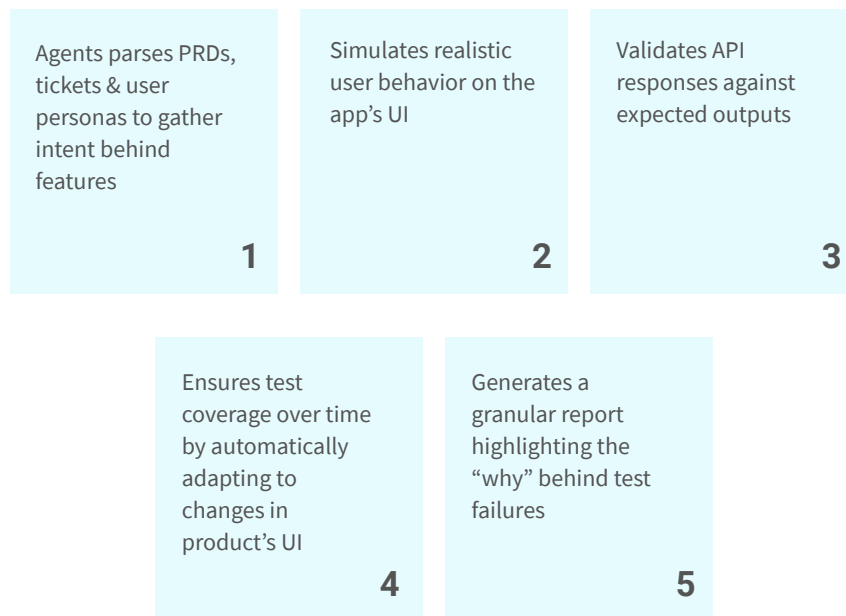
2. Functional Test Agents

Problem

QA teams invest significant manual effort in identifying user workflows, crafting test steps, and executing them before each release. Also, implementing the same tests across multiple devices and firmware versions adds to the delay. These tests are often brittle, break with minor product changes, and quickly become stale. As a result, test coverage remains low, and teams often rely on customer-reported issues to catch regressions. Functional testing is currently a lagging process, slowing down releases, sub-optimal customer experience, and failing to scale with fast-moving development cycles.

Solution

There's a high-leverage opportunity to build agentic AI systems that act like autonomous QA analysts. These agents would continuously generate and run end-to-end functional tests without human intervention. For instance, we are seeing one of our portfolio companies, Drizz:



These agents function like tireless QA engineers embedded into the CI/CD pipeline, catching regressions early, enhancing test depth, and eliminating manual test maintenance.

Given the nascent market adoption, the trust over AI executing functional tests without human oversight will be built over time, starting with shadow mode and moving to supervised mode in the near term and autonomous mode in the long term.

Business Characteristics of Functional Test Agents

Type of AI	Agents
Type of work	Execution + Creative Execution - Running the test cases across devices through visual and textual inputs Creative - Generating user flows and breaking them down to smaller steps to design test cases
Scope	Due to high engineering complexity and the creative nature of the opportunity, it is better to approach product development with a limited scope to ensure highly accurate test coverage.
ICP	Senior Engineer/PM
Technical feasibility	High Complexity Multi-modal context: Agents must deeply understand the application by ingesting PRDs, code, UI structure, backend systems, user flows & business logic, needing advanced engineering. Long-horizon memory: User flows can span 20–100 steps. Maintaining accuracy across such sequences demands robust long-term memory. UI resilience: Scripts should adapt to UI changes (e.g., label shifts, button reorders) via intelligent selectors and visual reasoning. UI automation: Simulating real user behavior across browsers and devices is hard. Tools like Playwright help, but adaptive, pixel-level execution remains complex. Simulation infra: Running UI tests at scale requires a device farm, building in-house is tough, and outsourcing (e.g., to BrowserStack, AWS) adds to costs. CI/CD integration: Agents must plug into pipelines (GitHub Actions, Jenkins, etc.), manage artifacts, and run consistently in containerized setups. Feedback loops: Diagnosing failures and evolving test cases needs agents that learn from test history and production signals, requiring retrieval-augmented memory or fine-tuning.
Ease of GTM	Medium complexity: Outbound motion with solution engineering support in early sales. QA managers are likely internal champions, but may lack budget ownership. Founders will need buy-in from engineering leadership. Pilots must show real reductions in manual test writing & failure debugging. Mid-funnel driven sales i.e., strong demo environments will be key. Integration Overhead: Functional testing touches frontend UIs, APIs, test data & CI/CD pipelines. Deep integration is needed with tools (Jira, GitHub, & test env) which create friction. Time-to-Value: Early results may take weeks as the system needs to ingest PRDs, existing test cases, and understand workflows – requires a clear ROI story to keep prospects engaged.
Market size	Functional testing makes up 50–60% of QA budgets in enterprise teams. Mid-sized SaaS companies spend \$500K–\$2M annually on QA headcount, tools, and infra. Test management and automation tools (e.g., Selenium grid, BrowserStack) can cost \$50K–\$200K per year in larger orgs. AI Agents could displace both outsourcing (QA vendors) and internal headcount by delivering immediate ROI on cost and speed, unlocking access to services' share within IT budgets. The global software testing market is ~\$40B+, with functional testing the biggest slice. Even focusing only on mid-market SaaS & digital-native enterprises, the SAM is \$3–5B+.
Defensibility	Tribal Knowledge: The agent learns each app's quirks (preferred paths, common bugs, and domain logic), improving over time by collecting flows, failures & regressions across releases. Structural Commitment: As agents replace in-house QAs, they simplify operations and establish new QA workflows and habits. Trust Loops: Accurate, consistent catches build user trust. Over time, this self-reinforcing feedback loop becomes a key moat. Integrations: Deep ties with CI/CD pipelines, test runners, and staging environments create high switching costs, embedding the agent in the release cycle.

Landscape



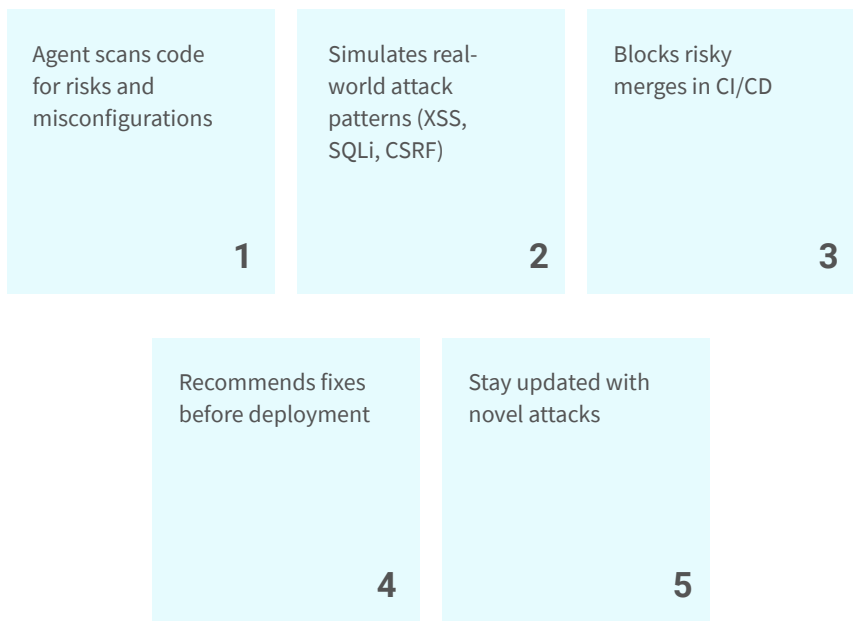
3. Security Testers (Shift-Left Security)

Problem

Before deployment, code must pass security checks to meet compliance and governance requirements, aimed at reducing data and access breach risks. Currently, security testing is late, reactive, and disconnected from daily development. Also, current tools miss zero-days and novel exploits because they follow standardized, static rules.

Solution

AI agents that act like embedded red teams, testing security during development, shifting the testing process left.



This AI tests continuously, not just during audits.

Business Characteristics of Security Testers (Shift-Left Security)

Type of AI	Copilot + Agents Copilot will detect vulnerabilities, potential impact and suggest changes to the code Agents create test scenarios and attack the system
Type of work	Creative (threat modeling) + Execution (exploit attempts)
Scope	Narrow , due to high engineering & GTM complexity. Advisable to pick a vertical and be an expert in it.
ICP	Senior security engineers in the CISO's organisation
Technical feasibility	Extremely high complexity Vulnerability Detection: Requires training/fine-tuning models on security-specific corpora, like vulnerable databases (e.g., CVE, OWASP), bug bounty reports, code exploits. Not easily done with generic LLMs. Attack Simulation: Needs an agentic model that can emulate attacker behavior, manipulate API inputs, mutate tokens, bypass auth flows, etc. Difficult and dangerous to get right. Context Awareness: Must understand the full software stack, including auth layers, session handling, business logic – to properly simulate threats. False Positive Handling: Security teams don't tolerate noisy tools. The AI must be highly precise and support explainability (e.g., "this token is vulnerable due to X pattern"). Updated Attacks: Keeping the library of security issues updated will be challenging. Partner with companies that maintain the updated list of attacks and their patches for their vertical.
Ease of GTM	Complex Top-down Motion: You'll need outbound sales, credibility, and partner integrations (e.g., with Splunk, Palo Alto, HashiCorp, etc.) to drive adoption. The buyer is more senior (CISO, security engineer, compliance lead). This means procurement hurdles, longer sales cycles, security reviews, and proof-of-compliance processes. Consider partnering with security consultancies or compliance platforms to drive wedge entry. Time-to-Value: Proving value requires running simulations and identifying novel vulnerabilities, potentially high impact, but takes time to set up and contextualize. Competitive Risk: Innovation-first players in overlapping spaces such as static application security testing (SAST), Dynamic analysis (DAST), and DevSecOps platforms (e.g., Snyk, Checkmarx) will add challenges to the sales process.
Market size	Security budgets in enterprise IT are rapidly growing, with AppSec alone projected to hit \$7–10B globally in the next 3–5 years. Large orgs spend \$500K–\$5M/year on AppSec depending on the sector (fintech, healthcare, e-commerce). AI Security Agents could replace static scanners and manual pen-test cycles, offering continuous security validation integrated into CI/CD. The shift-left security tooling market is already valued at \$3–4B. AI-native tools targeting this space could realistically unlock \$500M–\$1B+ in the near term.
Defensibility	Simulation and Fuzzing Engine: Developing a robust, proprietary sandbox to simulate attacks in context (not just static scanning) is a major technical moat. Tribal Knowledge: Encoding red-team tactics, threat modeling frameworks, and domain-specific risks (e.g., fintech vs. healthcare) into the agent creates specialized defensive layers. Compliance Loops: Auto-tagging vulnerabilities to compliance frameworks (SOC2, HIPAA, PCI-DSS) adds defensibility via regulatory alignment – critical for enterprise sales. Enterprise Integration Depth: Integrations with SIEMs, ticketing systems (e.g., Jira), identity providers (e.g., Okta), and code platforms make replacement painful and costly.

Landscape      FireCompass 



5

Deployment

Deployment remains reliant on ad-hoc setups and implicit know-how, despite automation through IaC and CI/CD tools. AI changes this by learning from workflows and telemetry to suggest or generate optimized, secure pipelines. Agentic systems go further, enabling intent-driven deployments with minimal human input.

Once code is written and tested, deploying it to production is a critical step that involves configuring infrastructure, setting up build and release pipelines, managing environments, and ensuring scalability, availability, and security.

Traditionally, this began with engineers provisioning cloud servers manually via the consoles of cloud providers like AWS or Azure. With the rise of Infrastructure as Code (IaC) tools such as Terraform, Pulumi, and AWS CloudFormation, teams gained more repeatability and automation. But IaC, while powerful, is essentially declarative, it exposes switches and dials, but the architecture and operational flow must still be designed and configured manually.

Today, engineers work closely with DevOps or platform teams to build CI/CD pipelines using frameworks like GitHub Actions, CircleCI, ArgoCD, or Spinnaker. This involves a range of tasks: containerizing applications, defining deployment targets, writing pipeline logic, and monitoring rollouts. Much of this work is boilerplate or based on well-understood patterns, but small mistakes can cause outages or security lapses, so it's treated with extreme caution. In many organizations, the deployment process becomes tribal, a set of scripts, conventions, and practices passed down over time, without formal documentation or reusable abstractions. As a result, it is both time-consuming and prone to drift.

What changes with AI

AI can transform deployment by learning from existing CI/CD workflows, application telemetry, system bottlenecks, and infrastructure configurations to suggest or even generate deployment pipelines that are secure, scalable, and cost-effective. For instance, a model trained on hundreds of thousands of deployment patterns could recommend whether to use blue-green or canary deployments for a new microservice based on its user behavior and update frequency. It could spot insecure configurations, suggest resource allocations based on usage forecasts, or optimize container orchestration strategies.

Agentic workflows can go one step further: rather than simply offering suggestions, an agent could orchestrate the entire deployment process. Given a high-level intent (e.g., “deploy this Python service to production with autoscaling and zero downtime”), an AI agent could generate and apply Terraform code, configure Kubernetes manifests, set up observability hooks, and push updates through a staging pipeline, with human approvals in the loop. This unlocks a new level of abstraction where infrastructure decisions are not hardcoded but are contextually adaptive.

Opportunities for startups

1. AI Copilot for Deployment

Problem

The core problem is the cognitive overload developers face when making infrastructure decisions across dozens of tools and settings. Developers and DevOps engineers spend a significant amount of time deciding how to deploy applications, like choosing the right strategies, configuring environments, managing secrets, and writing YAML or JSON configuration files. These decisions are complex, tribal, and hard to document. Even small missteps can lead to outages or security breaches, so teams move slowly and cautiously, often reinventing the wheel for each new service.

Solution

An AI-powered deployment copilot that can act as a reasoning engine to suggest or auto-generate optimized deployment plans.

Copilot synthesizes best practices, historical configs, infras state & telemetry data to suggest optimized deployment plan

1


Helps engineers choose the right patterns, write clean IaC, validate configs & flag risky setups while explaining the trade-offs

2

Rather than executing changes by itself, it augments engineer's decision-making with intelligence & guardrails

3

Business Characteristics of AI Copilot for Deployment

Type of AI	Copilot
Type of work	Creative It supports reasoning, comparison, and planning rather than repeatable actions.
Scope	Start with narrow and go wide Pick a vertical (eg. deploying ML models) to get started. Later, span across IaC, CI/CD, monitoring hooks, deployment strategies, and rollback configurations.
ICP	Platform engineers, DevOps teams, and tech leads
Technical feasibility	High , as existing foundation models can be fine-tuned with open-source infra patterns and IaC templates.
Ease of GTM	Medium ; can be distributed via plugins to GitHub/GitLab or embedded in CI/CD dashboards.
Market size	Large ; the CI/CD and IaC ecosystem is already a multi-billion-dollar market. However, market size may be constrained if positioned as a standalone product, it works better as a wedge into broader developer experience platforms.
Defensibility	Medium ; defensible via proprietary telemetry data, usage context, and integration depth.
Landscape	

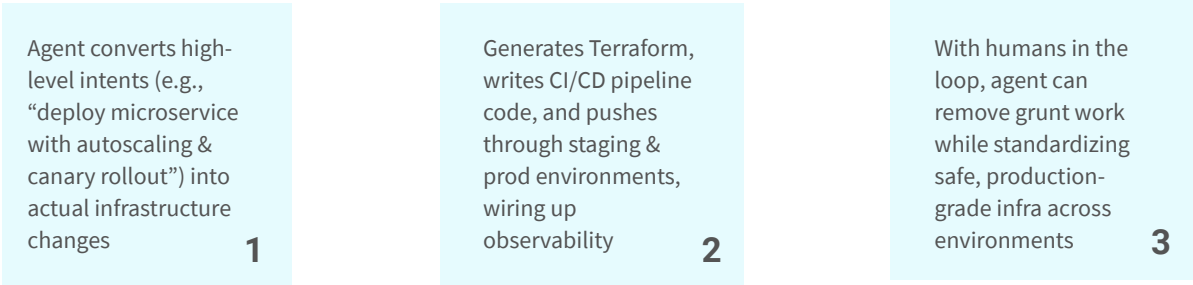
2. End-to-End Deployment Agent

Problem


The operational burden of implementing and modifying CI/CD pipelines, configuring cloud environments, and provisioning infra is heavy. Platform teams spend cycles on writing YAML pipelines, managing state files, rolling back deployments, configuring observability and alerts, most of which follow known patterns. Yet every team still does this manually. This work is tedious, time-consuming, and prone to configuration drift or security missteps.

Solution

An AI deployment agent can convert high-level intents into actual infrastructure changes.



Business Characteristics of End-to-End Deployment Agent

Type of AI	Agentic
Type of work	Execution It automates routine setup and maintenance tasks rather than making architectural decisions.
Scope	Narrow It must start with well-scoped stacks (e.g., Node.js + K8s on AWS) due to diversity of infra.
ICP	Internal platform teams, DevOps consultancies, CTOs of mid-market companies, and developer tool teams
Technical feasibility	Medium ; while it is feasible for standardized use cases, but harder to generalize across stacks and orgs. Even with accurate generation, the risk of deploying incorrect or insecure infrastructure is high. Moreover, the agent must interface with a wide range of APIs, permissions, and stateful systems, making it hard to generalize.
Ease of GTM	Low ; requires high trust, security reviews, and likely a high-touch sales or onboarding process. From a GTM perspective, this would likely need a service-heavy onboarding, perhaps targeting digital-native enterprises with dedicated internal platform teams.
Market size	Medium ; valuable to existing CI/CD players and enterprises, but hard to penetrate as a standalone tool. The market size is significant (infrastructure automation is a multibillion-dollar category) but monetization will depend on trust, reliability, and compliance guarantees. Unlike copilots, agents have a thinner value proposition unless they replace entire DevOps teams, which is unlikely in the short term.
Defensibility	Low ; unless paired with deep integrations and proprietary telemetry or policy enforcement frameworks. Defensibility will hinge on proprietary datasets (e.g., internal infra blueprints), agent orchestration frameworks, and integration depth with CI/CD tooling.
Landscape	

Deployment, like system design, sits at the high-complexity, high-risk end of the SDLC. While AI copilots can augment developers with decision support and configuration suggestions, agentic execution faces greater hurdles due to the fragmented nature of infrastructure, the high bar for reliability, and the low tolerance for errors. We believe deployment copilots will emerge faster and with clearer ROI than full agents. Over time, as trust and verification layers improve, agentic workflows may evolve to handle more execution, but will likely remain bounded to specific domains or internal tooling. The larger opportunity may lie in embedding intelligence into existing CI/CD providers rather than reinventing the deployment stack from scratch.



Maintenance

AI is reinventing software maintenance by going beyond alerting and ticket routing. It enables root-cause analysis, automated remediation, contextual onboarding, and smart ticket resolution, cutting manual effort, improving reliability, and turning monitoring and support into adaptive workflows that evolve with usage.

The maintenance phase of the SDLC ensures the system runs reliably after features are shipped. Unlike development phases that “complete,” maintenance never ends. It includes two major workflows: application monitoring and customer support.

Application Monitoring

Modern applications use observability platforms (Datadog, New Relic, Grafana, etc.) that collect logs, traces, and metrics. These tools analyze real-time signals and notify developers when something breaks or drifts out of baseline. Most tools are good at answering: “Is something broken?”

While useful, they mostly operate at a symptom level. Alerts are triggered when anomalies are detected, often without clear root causes or resolution steps. Engineers must still:

- Reconstruct the timeline of events
- Correlate across metrics and logs
- Identify the root cause
- Apply a fix

Current APM tools are great at detection but fail at explanation and remediation. It’s like having a smoke alarm, but no fire inspector or sprinkler.

Customer Support

There are two segments to customer support: the user onboarding and ticket handling

User Onboarding: Once a product is deployed, users must learn how to extract value from it. This is especially challenging in the age of AI-native software, where products often require a change in user behavior or process. As a result, users frequently get stuck mid-workflow, leading to frustration and delayed outcomes.

Early platforms like Whatfix and WalkMe attempted to address this by enabling SaaS companies to build in-app walkthroughs and onboarding flows. However, these systems resemble static test cases: fragile, short-lived, and rarely comprehensive. Maintaining coverage across evolving UIs or feature sets becomes a manual, time-consuming task, limiting their effectiveness at scale.

Ticket Handling: When users experience bugs, support tickets are created and routed to engineering via platforms like Jira, Zendesk, or Linear. Over 50% of these tickets in internal tools are duplicates or have known fixes. While the resolution workflows are still human-heavy:

- Engineers read, reproduce, and resolve the issue
- Time is spent deciphering vague ticket language
- Context switching reduces developer velocity

Today, even the most “intelligent” ticketing tools simply tag, prioritize, and route. The actual debugging and resolution remain manual.

What changes with AI

AI has the potential to fundamentally reshape both monitoring and support, moving us from passive alerting to active remediation.

Application Monitoring

AI systems can now ingest and reason across structured observability data, like logs, traces, metrics, and combine it with code context (e.g., recent diffs or deploys), historical incidents and resolutions, system topology (what depends on what), and git history (when and where changes happened). This allows AI to answer a new progression of questions, like “What broke?”, “Why did it break?”, or “Can you fix it?”

AI enables two core capabilities. First, RCA copilots, which provide likely root causes, reasoning paths, and suggested actions in plain English (e.g., “500 errors spiked after deployment X. Likely culprit: unhandled edge case in API Y”). The second is remediation agents. For recurring or low-risk issues, agents can trigger automated rollbacks, restart services, tweak configurations, or even raise pull requests for review. This shifts monitoring from alert-and-react to observe-and-adapt.

Customer Support

AI is transforming both user onboarding and ticket handling from static, reactive processes into dynamic, proactive workflows that enhance user success and reduce friction in support.

In the case of user onboarding, AI enables conversational and context-aware guidance that replaces brittle step-by-step walkthroughs with a smarter, more responsive training layer. Contextual assistance comes in the form of chatbots or in-app copilot overlays that can detect where users are stuck within a workflow and offer on-the-spot help through tooltips, examples, or guided explanations. Adaptive learning allows AI to dynamically generate or surface micro-tutorials tailored to the user's pain points based on usage patterns and errors. With continuous improvement, AI can auto-adjust help content as features evolve (or flag outdated walkthroughs) and even suggest new ones. This flips onboarding from a static directory of how-to's to a live, personalized companion that meets users where they are.

For ticket handling, AI is shifting the process from manual triage to automatic inspection and resolution. AI-powered systems are now capable of semantic understanding, using NLP to extract intent and context from user messages and link them to similar past tickets or known bug patterns. Through automatic triage and repair, AI agents can auto-suggest patches, configuration tweaks, or workarounds for well-known issues, and even generate draft PRs. To close this loop, AI can produce clear, personalized responses that detail the fix, root cause, and preventive advice for the user. As a result, ticketing tools are evolving from basic ticket routers into semi-autonomous support agents that handle the mundane and escalate only the complex.

Opportunities for startups

1. AI SRE

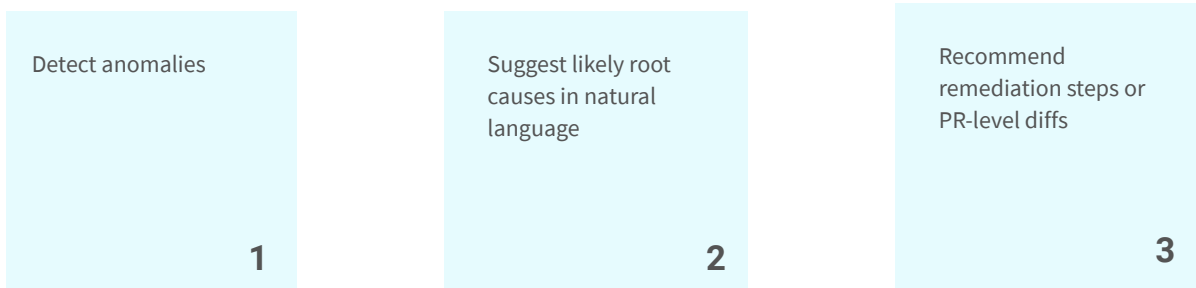
Problem

Today’s observability tools surface what broke, but not why. Engineers still spend hours reconstructing timelines, sifting through noisy logs, and correlating signals to pinpoint issues, especially in microservice or distributed environments.

Once the RCA report is generated, fixes outside business logic are straightforward and often repetitive. Engineers manually resolve the same issues: restart services, revert bad configs, tweak infra. These playbooks are known but still require human execution.


Solution

An AI copilot that connects logs, traces, metrics, recent commits, and deploy metadata.



On top of this, an agentic platform can map known alerts or and autonomously execute safe fixes like restarting pods or rolling back deployments. It involves humans only when confidence is low.

Business Characteristics of AI SRE

Type of AI	Copilot + Agentic A combination of copilots for reasoning and suggesting RCA, and agents for automated execution of known fixes.
Type of work	Creative + Execution The initial phase of identifying root causes across distributed systems is creative and requires contextual reasoning and planning. Once the RCA is established, the remediation phase is largely execution-driven, following predictable patterns and predefined runbooks.
Scope	Wide; the solution must operate across a wide range of observability stacks, infrastructure layers, and logging formats. It should be adaptable to different environments: cloud-native systems, Kubernetes clusters, and even monolithic architectures.
ICP	Site Reliability Engineering (SRE) teams, DevOps engineers, and platform teams in mid-to-large digital-native enterprises.
Technical feasibility	Extremely Difficult Training Data Scarcity: High-quality, structured training data (logs, traces, metrics) is not publicly available. This data is proprietary and tightly coupled with the company's core application infrastructure making access a significant challenge. Customization Needs: RCA workflows vary across industries, engineering cultures, and org structures. Building generalizable models that capture these nuances while delivering accurate outputs is a deeply technical undertaking. Accuracy Expectations: Dev leaders have a very low tolerance for false positives or partial results. If the system works in only 70% of cases, developers will still need to manually verify every output, negating the value of automation. Fragmented Tooling: Most large organizations use 20+ observability, deployment, and alerting tools. Normalizing and correlating data across these disparate sources is a highly non-trivial problem.
Ease of GTM	Hard; the product will require a top-down go-to-market motion, as the ICP is limited to scaled companies with high operational maturity and a focus on uptime SLAs. Gaining access to production systems and observability pipelines requires significant trust, compliance alignment, and security clearance – difficult hurdles for early-stage startups. That said, the value proposition is extremely compelling: weeks of engineering effort saved per incident. Early adopters willing to experiment with emerging tech may serve as foundational design partners. A phased approach is advisable; start with a copilot for RCA, and expand into agentic remediation over time.
Market size	Large; every cloud-native company with microservices spends heavily on observability. TAM includes monitoring, incident management, and ops automation (multi-billion dollar space).
Defensibility	As of today, the number one defensibility is figuring out a working product that gives entry into customers & log usage of the devs. Access to high-quality, private operational data will compound to become a durable moat. Deep integrations with proprietary observability stacks and internal incident workflows create strong switching costs. The potential to build domain-specific knowledge graphs that improve over time adds product value.
Landscape	

2. AI Onboarding Copilot for complex SaaS workflows

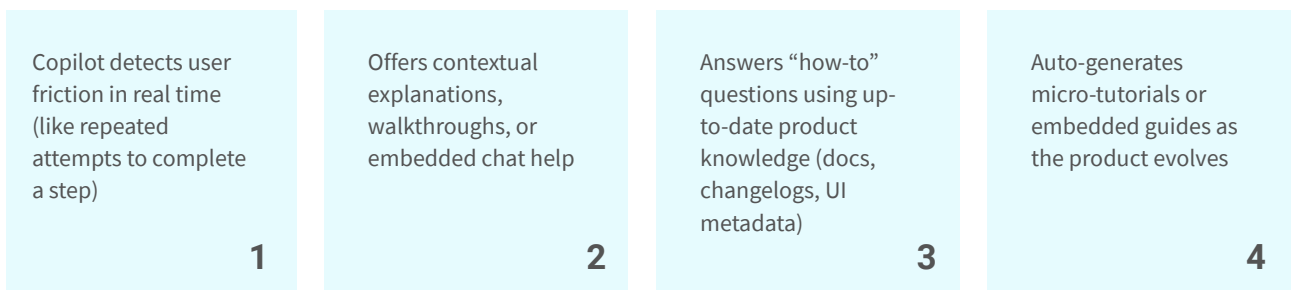
Problem

As SaaS products grow more powerful (especially AI-native apps), they also grow more complex. Users often struggle to complete onboarding, configure the product, or understand how to extract value. This creates a high volume of support queries that are not bugs, but “how do I use this feature?” questions.

These queries account for a disproportionate load on support and CS teams and slow down time-to-value (TTV) for customers.

Solution

An in-app, LLM-powered onboarding copilot that follows the below workflow.



It acts as a smart CS assistant baked into the app, handling intent-aware guidance without human involvement.

Business Characteristics of AI Onboarding Copilot for complex SaaS workflows

Type of AI	Copilot Primarily a copilot that augments users during onboarding. It detects friction, surfaces relevant help content, and provides in-context guidance via chat, tooltips, and walkthroughs.
Type of work	Creative While not writing code, the system must understand user intent and offer the right guidance at the right time, which requires planning, reasoning, and interpretation of ambiguous signals. It's about assisting learning and problem-solving rather than task execution.
Scope	Wide. Although the solution must be embedded deeply into each product's UI and workflows, the platform itself is horizontal. It must generalize across SaaS products, interpreting varied UI components, detecting diverse friction patterns, and adapting to new workflows. A strong foundation in user intent modeling and UI analysis is key.
ICP	Product managers, growth teams, and customer success leaders at B2B SaaS companies especially those with complex user workflows or AI-native products that require behavior change or deep configuration.
Technical feasibility	Medium-High Data Availability: Internal access to user behavior, UI metadata, documentation, and telemetry is typically available making training and inference viable. Model Complexity: Requires strong retrieval systems (for docs, changelogs, FAQs), LLM orchestration, and lightweight personalization based on real-time product usage. Product Integration: Needs tight embedding within the application (e.g., tracking flows, UI events), but technically manageable with SDKs or browser overlays. Workflow Mapping: Translating documentation (PDF, Text, Image) into workflow maps (data structures like linked lists, trees) will be a technically complex job that the product needs to automate or find accelerated solutions. Also, removing outdated workflows and updating newer ones will require deep integrations with the production dev cycle.
Ease of GTM	Medium Pros: Clear ROI through reduced ticket volume and improved product activation. Can be bundled with CS or onboarding tools already used by SaaS teams. Challenges: Not bottom-up; requires coordination with product and CS leadership. Initial implementation can be non-trivial, but full onboarding coverage requires mapping workflows & edge cases. TTV (Time-to-Value) may be longer than typical CS tools due to integration complexity.
Market size	Medium While every B2B SaaS company needs onboarding, only a subset prioritizes real-time, adaptive onboarding today. Growing demand as SaaS products become more complex and lean teams need scalable customer education. Potential to expand into adjacent markets (product tours, feature discovery, sales, or Q&A).
Defensibility	Strengths lie in proprietary telemetry and user behavior data, which create compounding model value over time. Deep integration with real-time events and UX context builds strong switching costs. The key risk: generic LLM + RAG approaches risk commoditization unless tightly enriched with product-specific signals. Defensibility improves when behavioral feedback loops are closely tied to output quality.

Landscape  Adopt  Whatfix  stonly  Optextity  apty

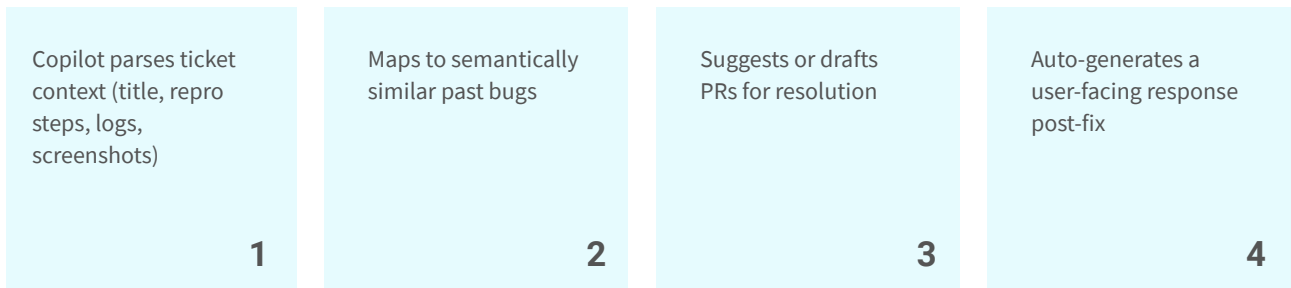
3. Support Ticket Resolution Bot

Problem

Support and internal dev tools generate hundreds of recurring tickets. Many follow known patterns and resolutions, yet still require human review, triage, and patching.

Solution

An in-app, LLM-powered agent that follows the below workflow.



Business Characteristics of Support Ticket Resolution Bot

Type of AI	Agentic
Type of work	Execution The core job is to understand problem descriptions, identify known issue patterns, and apply standardized fixes (tweaks, patches, PRs). While some semantic understanding is needed to parse vague tickets, the work itself is procedural and repetitive, ideal for automation.
Scope	Wide The bot must generalize across support systems (e.g., Zendesk, Jira, Linear), bug types, and communication styles. It also needs to interface with engineering workflows (e.g., GitHub, CI/CD pipelines), making it relevant across many domains and technical stacks.
ICP	Support operations teams, internal tools teams, and IT/helpdesk teams at mid-to-large tech teams, especially those with high & frequent support ticket volume/large internal platforms.
Technical feasibility	Medium Data Availability: Most companies have access to historical ticket data, logs, and code repositories. However, this data is often unstructured, inconsistently formatted, and spread across multiple systems, making preprocessing and normalization a significant challenge. Model Complexity: The system must semantically understand vague or incomplete problem statements, identify similarities with past issues, and map them to the appropriate resolution steps. Additionally, it must estimate confidence levels for proposed fixes to determine when human intervention is necessary. Integration Requirements: Requires deep integration with ticketing systems (for ingestion), version control systems (for suggesting or drafting fixes), and deployment pipelines (for pushing changes or triggering rollbacks). The complexity of these integrations increases in enterprise environments with custom workflows. Product loops: To be accepted in production workflows, the system must deliver high accuracy & reliability, and be designed to fit within existing team processes, ensuring human verification and sign-off before any automated PRs are merged or changes are pushed.
Ease of GTM	Medium Trust Barrier: Support leaders are risk-averse and will not automate ticket resolutions without a high degree of confidence and explainability. A single incorrect or “hallucinated” fix can erode trust and introduce operational risk. Adoption Path: Likely requires a top-down, land-and-expand motion, starting with internal tools, non-critical ticket categories, or sandboxed environments. These early deployments can serve as proof points to demonstrate ROI and build internal credibility. Value Proposition: Once trust is established, the payoff is significant, dramatically reducing response times, resolution effort, and overall support costs.
Market size	Large. Every tech-driven company handles internal or external support tickets, with growing volume as systems and user bases scale. Applicable to both external customer support and internal IT/engineering ops (e.g., platform support, CI/CD issues). Expansion potential into ITSM automation, chatbot-assisted support, or even full-service AI helpdesk platforms.
Defensibility	Data moat: Access to historical tickets, private bug data, and code integration will create a long-term edge. Similarly, building a knowledge graph of issue-resolution mappings that improve over time will be a powerful advantage. Product loop: The taste of high accuracy and the habit to adopt the solution will be an effective strategy to continue winning in the market.

Landscape  DevRev  Forethought  SysAid  THENA  okapture

Closing Note

Through this report, we've journeyed through every phase of the software development lifecycle, not just to admire the power of AI, but to understand its precise leverage points and limitations across ideation, execution, and scale.

What we've seen is clear: AI is no longer a shiny add-on. It is quietly becoming the scaffolding behind how modern software is envisioned, developed, tested, shipped, and modified. From copilots that accelerate creativity to agents that handle mechanical grunt work, the development process is moving from human-powered to human-directed.

But this transformation isn't uniform. Some domains, such as code generation and frontend scaffolding, are getting real adoption, while others, like security testing and system design, remain early-stage, constrained by trust and training data. Though horizontal copilots have drawn widespread attention (and some skepticism), it's the focused, embedded agents that are driving meaningful productivity and sustainable competitive advantage.

Below is a closer look at the level of bullishness of the 15 opportunities at Stellaris for potential investments:

Most bullish today (Have invested and actively looking for companies to invest in)	Frontend Execution Agent, Functional Test Agents Specialised coding agents AI Onboarding Copilot for Complex SaaS Workflows
Hard to differentiate from competition (Observing the space closely; massive opportunities to partner with the right team)	AI Designer Assistant, Codebase navigator Code Change Impact Analyzers AI SRE, Zero-Code App Builder
Long-term opportunities (Technically infeasible at the moment)	System Design thinker, AI Copilot for Deployment Support Ticket Resolution Bot End-to-End Deployment Agent System Design executor, Security Testers

If there's one big shift underway, it's this: software development is moving from an artisanal craft to a systems design problem. Just like cloud turned infrastructure into APIs, AI is turning software engineering into chat-based orchestration. And in that world, the startup opportunities are immense, not in replacing human engineers, but in elevating their impact.

The next GitHub or Atlassian won't be a prettier IDE or a faster CI tool. It will be the invisible layer that blends reasoning and execution across the lifecycle, turning "work" into workflows and code into conversations.

As AI becomes the default teammate in every engineering organization, the real question is no longer *whether* it will transform the SDLC. It's *who* will build the best interfaces, infrastructure, and intelligence to get us there. If you're building in this space, write to us at stellarisvp.com/team.

Write to us at stellarisvp.com