

Opening illustration. AI-generated conceptual artwork for LMM Labs.

Lee Farretta and Matthew Farretta • LMM Labs

Prepared in collaboration with ChatGPT and Codex

The Anatomy of an Accelerator

A design for the LMM Labs general-purpose AI application accelerator

Abstract

LMM Labs proposes an application accelerator that can carry engineering decisions from one software project to the next. The proposal draws on earlier work in visual development, software product lines, and application generators, as well as our experience directing coding agents across a portfolio of applications. We describe the parts of the proposed platform and the relationships that give them a common purpose. A catalog of application patterns supplies reusable requirements and tests. A project brief records the choices specific to one product. AI-driven orchestration uses both to plan construction and revise the next step in light of build and test results. Human reviewers retain authority over product behavior and release. We explain how this architecture can serve several application families as modules are added, and how a first prototype could establish whether it reduces the human effort required to deliver working software.

Introduction

An application accelerator is a software platform that helps a team turn a product idea into a working application by reusing designs, components, and tests across projects. A team building a new product still has to decide what its users need. The opportunity for acceleration lies in recognizing where an earlier project has already settled an engineering question and making that answer available when the team needs it again.

At LMM Labs, we came to this idea through our own development work. Our October case study documented 14 applications, including 12 with evidence of cloud deployment (Farretta & Farretta, 2026). Our subsequent working count reached roughly 18 applications in 30 days. Codex performed much of the detailed implementation, while we defined each product, directed the work, and judged the result. As the portfolio grew, a recurring demand on our time became apparent. Even after we had settled a requirement, someone still had to carry that decision into the next coding assignment, the tests, and the release.

We propose to build a general-purpose AI application accelerator that takes on more of that coordination. This article describes its anatomy: how a catalog of familiar application patterns informs the design of a particular product, how the platform directs a coding agent from that design, and how reviewers determine whether the delivered software meets the original brief. The proposal draws on a long history of development tools. Understanding what those tools made reusable helps explain where AI might extend their reach.

A history of rapid application development and application accelerators

One early lineage of application acceleration began with a familiar business problem. Companies needed many different reports, yet the programs that produced them repeatedly performed the same operations. In 1959, IBM described a Report Program Generator for its 1401 computer. A programmer specified the input record layout, calculations, and printed output, and the generator produced a program tailored to that report (IBM, 1959). By 1977, Zloof and de Jong had extended this idea in their experimental System for Business Automation. People who understood a business task could demonstrate operations on tables and forms and use those examples to define an application. In both systems, the tool could take over detailed programming because its designers had given it a precise way to represent a familiar class of work.

Rapid application development (RAD) broadened the question from how to generate a program to how quickly users and developers could agree on the right one. In *Rapid Application Development*, Martin (1991) brought together joint design workshops, working prototypes, and fixed development intervals. Users could examine an early version while there was still time to change it. Visual Basic gave that approach a practical tool on the desktop: developers could assemble an interface from reusable controls and write the behavior specific to the application (Hsu, 2017). The two advances addressed different sources of delay. Reusable controls reduced construction work, while early review reduced the time a team could spend building to a mistaken assumption.

In the 2000s, software factories made the application family a unit of reuse. Greenfield and Short (2004) described combining models, patterns, and tools to assemble related products. Software product line engineering formalized which assets a family could share and which decisions belonged to each product (Northrop et al., 2012). Rails scaffolding put that principle to work for web developers by generating a model, routes, views, and tests for a resource (Ruby on Rails, n.d.). By 2021, VMware's Tanzu Application Accelerator offered teams project templates that incorporated their organization's code and configuration standards (Bock & Manachi, 2021). A coding agent can now modify the resulting project and run checks against those changes (OpenAI, 2025). For LMM Labs, this raises an architectural question: what parts must a platform have to direct that work from an approved requirement through a verified release?

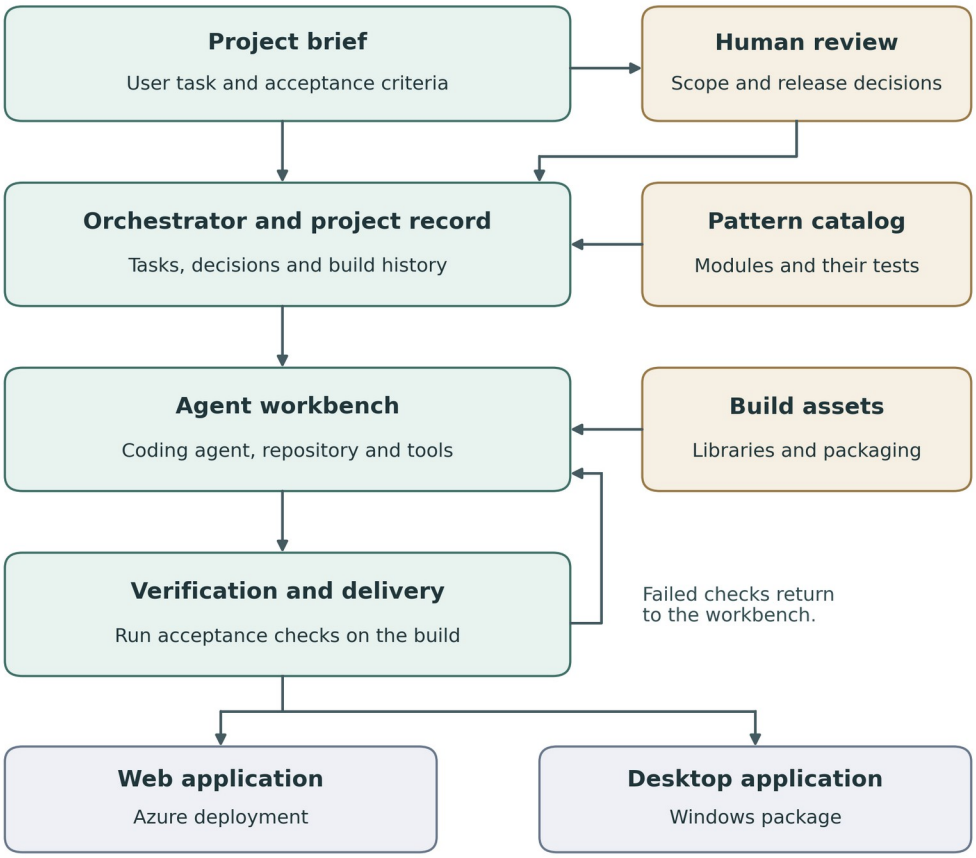
The proposed anatomy of the LMM Labs accelerator

In the proposed LMM Labs accelerator, a team would define what an application must do and entrust the platform with coordinating the work needed to build and test it. The design must account for two kinds of knowledge. Some requirements recur across applications and can be kept in a shared catalog of patterns. Others reflect choices the team makes for one product. The platform needs to carry both into the instructions it gives the coding agent and the tests it uses to evaluate the result.

Figure 1 shows how the components work together. A project brief records what the team has agreed to build. The orchestrator uses the brief and the pattern catalog to assign work, and the agent workbench gives the coding agent the tools to carry it out. Verification checks the resulting application against the brief. The project record preserves the decisions and results so reviewers can see what passed, what needs repair, and what still requires their judgment.

Figure 1

Proposed Architecture of the LMM Labs AI Application Accelerator



Note. The pattern catalog supplies design choices and acceptance checks. The orchestrator records the choices accepted for each project. Failed checks return to the workbench, and a product decision returns to human review.

Across projects, the orchestrator follows the same basic process, but what it asks the coding agent to build and how it checks the result depend on the kind of application the team needs. The pattern catalog gives the orchestrator a starting point for both decisions by identifying requirements and tests that recur across similar applications.

The pattern catalog as an engineering foundation

For the accelerator to reuse engineering knowledge, it must first recognize which applications have a requirement in common. Two products may both run in a browser, yet one must reconcile a financial report while the other must prevent two customers from reserving the same seat. We use the term application pattern for a recurring user task, the engineering obligations it creates, and an observable test of success. The pattern catalog collects these descriptions so the platform can select a useful starting point for a new project. This approach draws on recurring models in domain analysis (Fowler, 1996) and on the practice of defining categories for an explicit engineering purpose (International Organization for Standardization, 2015).

Table 1 sets out eleven target patterns. Each catalog entry describes a family of applications. A module is the implemented part: it supplies design questions, reusable assets where appropriate, and checks the accelerator can execute. We estimate the incremental effort of the first reusable module in each family. A high score identifies costly design or verification work. The order in which products need those modules can differ from their effort ranking.

Table 1

Application Patterns Ranked by Estimated Module Development Effort

ID	Rank	Application pattern	Scores	Total	Principal effort driver
01	1	Record and collection management	1/1/1/1	4	Forms, validation, persistence
05	2	Analytics and reporting	2/1/1/2	6	Reconcile metrics to records
11	3	Content viewing and navigation	2/1/2/2	7	Viewer engine and session history
07	3	Authoring, editing, and publishing	2/2/1/2	7	Undo and export fidelity
04	5	Search, discovery, and retrieval	2/2/2/2	8	Index and relevance checks
02	6	Workflow and case management	2/3/2/2	9	Roles and durable handoffs
06	6	Modeling, planning, and simulation	3/2/1/3	9	Domain expertise and test cases
09	8	Data capture, transformation, and integration	2/2/3/3	10	Connectors and recovery
10	9	Monitoring, alerting, and event response	2/3/3/3	11	Late and duplicate events
03	10	Transactions and reservations	3/3/3/3	12	Concurrent claims and rollbacks
08	10	Communication and collaboration	3/3/3/3	12	Synchronization after reconnect

Note. ID is permanent. Pattern 11 is provisional. Scores run from 1 (low) to 3 (high) for implementation, state and concurrency, external integration, and verification, in that order. Equal totals share a rank. Estimates assume one delivery target and reuse of established components.

The catalog’s reach comes from two kinds of reuse. A single pattern can recur in several domains, and a single application can combine patterns. A travel service and a scheduling tool both face the obligation to prevent conflicting reservations. The same travel service may also need retrieval and monitoring. Each additional module must cover a distinct obligation. We would count an application as covered only if the selected modules account for its essential behavior. The proportion of applications that meet that standard remains an empirical question.

The project brief and application blueprint

Selecting a pattern narrows the design problem, but a team still has to decide how its particular application should behave. The project brief records who will use the product, what they must accomplish, and how the team will decide that it is ready. An approval workflow illustrates why that distinction matters. Requiring every designated reviewer to agree produces a different completion rule from allowing any one reviewer to approve. Both choices fit the same pattern (van der Aalst et al., 2003). The accelerator should bring that choice to the product owner and preserve the answer for implementation and testing.

The resulting application blueprint is the first project-specific output of the platform. It records the selected modules, the decisions that specialize them, the services or components they must share, and the checks that will demonstrate the combined behavior. Two individually sound modules can still make conflicting assumptions about ownership of a record or what happens after a failed operation. Those questions belong in the blueprint because they arise where the modules meet. A suite can use the same mechanism to specify behavior shared across several applications.

Orchestration and the project record

An approved blueprint describes the application a team wants. Building it may take several assignments, with a decision made in one affecting the work that follows. Orchestration is the platform function that coordinates those assignments. The orchestrator draws on a durable project record of requirements, decisions, dependencies, code revisions, and test results. In our current practice, a person often restores that context when an agent resumes work. With the proposed record, the platform could identify the last verified revision, the requirement still open, and the evidence needed to close it.

The project record also sets a limit on delegated authority. When an agreed check fails, the orchestrator can return the requirement and failure to the agent and request a repair. When two plausible interpretations of the requirement would give the user materially different outcomes, the next action is a product decision. The person who makes it should need to state it once. The orchestrator can then update the blueprint, identify affected work, and carry the decision through verification.

This is where the pattern catalog makes generality practical. The retrieval and modeling modules can ask different questions and supply different tests while relying on the same machinery to record answers, assign work, and evaluate results. The team can introduce either module when there is a real product to build. Extending the catalog demands engineering work, but it does not require a second orchestration platform.

Implementation, verification, and delivery

An assignment from the orchestrator sets a development task in motion. To complete it, the platform must give a coding agent the means to change the software and give a reviewer evidence about the resulting application. Implementation takes place in the agent workbench, where the agent receives the assignment, the relevant blueprint decisions, and a working copy of the code. The workbench supplies reusable components, configuration files, and packaging tools. It returns a specific revision and the results of its checks, so another agent or a human reviewer can pick up the work.

Verification then addresses the user's reason for wanting the application. A successful compilation establishes that a particular build step worked. An acceptance check establishes whether a user can perform an agreed task through the delivered product. The verification component therefore needs both the module's reusable checks and the project-specific conditions in the blueprint. It also needs an environment in which to run the result. For an Azure-hosted application, that means exercising its deployed entry point. For a downloadable Windows application, it includes installing and launching the package on a clean machine.

AI has a second, independent place in this architecture. A coding agent may have implemented the entire application, while the installed product may need a model for only one optional function. Our Windows browser illustrates the distinction: a user can browse normally and can supply a model API key to enable AI features. The blueprint must describe both operating conditions. The workbench provides the integration, and verification checks the user's experience with model access available and unavailable.

When reviewers accept the application, the delivery component links the approved blueprint, build revision, acceptance results, and deployable artifact. The team can then tell what it approved and reproduce that release. A later build may reveal a check worth adding to a shared module. Review matters at that point, too: a rule peculiar to one customer can become an expensive default if it enters the catalog as a general requirement. Reuse improves the next build only when the team transfers the right lesson.

What a first prototype could establish

The architecture rests on a practical claim: one platform can coordinate the development of different applications while reducing the effort people spend directing each build. Over the next few months, we propose to build and test a prototype against our current human-directed use of Codex. We would begin by building the smallest version that can accept a brief, retain a blueprint, assign a task, run a check, and resume after an interruption. A single coding agent is sufficient for this first test of whether the components work together.

Once the core platform is working, we propose to run repeated trials with two modules with materially different acceptance criteria. A Windows document retrieval tool must return evidence from a known collection and let a user reach the relevant passage. A web-based scenario modeler must reproduce independently calculated results when assumptions change. The pair tests whether the same platform can carry two kinds of engineering knowledge through two delivery paths. Because both pattern and delivery environment vary, the trials can demonstrate feasibility across that combination. Further trials would be needed to isolate the contribution of either factor.

To assess the results, we will compare the accelerator with people directing a coding agent on work of comparable scope. We will establish the brief and acceptance checks in advance, alternate which workflow goes first across the two task pairs, and record the assets each build inherits. The principal measure will be hands-on human time through an accepted release, including review, repair, and repeated instructions. Elapsed time, operating costs, unresolved defects, and the initial cost of building the platform will be reported alongside it. Becker et al. (2025) found that experienced developers using early-2025 AI tools took 19% longer on tasks in familiar repositories even though they believed the tools had saved time. Their setting differs from ours, but the gap between perceived and measured effort is directly relevant.

From AI-Assisted Coding to AI-Driven Orchestration

At LMM Labs, Codex has done much of the coding, while we have decided what to build, what needs attention next, and when an application is ready. We see AI-driven orchestration as the defining advance for the next generation of application accelerators. Earlier tools could carry out a sequence that developers specified in advance. An AI agent can also interpret what happened during a build and revise the sequence itself. That matters because every decision the agent can carry forward is one less point at which construction must wait for human direction.

Both researchers and platform developers are pursuing this shift. The ChatDev researchers demonstrated specialized agents coordinating design, coding, and testing through dialogue (Qian et al., 2024). GitHub's Agent HQ gives developers a common place to direct and track multiple coding agents (Daigle, 2025). OpenAI's Symphony delegates another part of the coordination to software by turning work items into agent runs (Kotliarskyi et al., 2026). Together, these efforts show how quickly orchestration has moved from a research question into the design of development platforms.

The opportunity for LMM Labs is to give that orchestration a practical foundation in application patterns. A platform that recognizes what different kinds of applications require can use that knowledge to guide work across projects. For a small team, the potential gain is substantial: more time to shape the product and judge its usefulness, and less time directing each step of construction. That is the promise our prototype will test.

Conclusion

The next step for LMM Labs is to build the accelerator and use it to develop two applications with different requirements. In the accompanying technical article, we will publish the prompts, interventions, failures, and results. That record should help us answer the practical question behind this proposal: can a small team spend less time directing construction and still deliver software it is prepared to release?

Research and preparation note

The paper draws on the published LMM Labs portfolio case study and the team's subsequent development experience. The October case study supplies the documented portfolio count at that date. The later count is the team's working estimate. The architecture and prototype trials are proposals. The module rankings express our judgment of relative engineering effort and will be revisited after implementation.

References

Becker, J., Rush, N., Barnes, E., & Rein, D. (2025). *Measuring the impact of early-2025 AI on experienced open-source developer productivity* [Preprint]. arXiv. https://arxiv.org/abs/2507.09089

Bock, B., & Manachi, R. (2021, September 1). *Announcing VMware Tanzu Application Platform: A better developer experience on any Kubernetes*. VMware Tanzu. https://blogs.vmware.com/tanzu/announcing-vmware-tanzu-application-platform/

Daigle, K. (2025, October 28). *Introducing Agent HQ: Any agent, any way you work*. GitHub. https://github.blog/news-insights/company-news/welcome-home-agents/

Farretta, L., & Farretta, M. (2026, October 4). *LMM Labs: An early-stage case study in human-orchestrated AI acceleration*. LMM Labs. https://lmmlabs.com/articles/lmm-labs-an-early-stage-case-study-in-human-orchestrated-ai-acceleration/

Fowler, M. (1996). *Analysis patterns: Reusable object models*. Addison-Wesley. https://martinfowler.com/books/ap.html

Greenfield, J., & Short, K. (2004). *Software factories: Assembling applications with patterns, models, frameworks, and tools*. Wiley. https://softwarefactories.com/TheBook.html

Hsu, H. (2017, April 11). *2017 CHM Fellow Alan Cooper: Father of Visual Basic*. Computer History Museum. https://computerhistory.org/blog/2017-chm-fellow-alan-cooper-father-of-visual-basic/

IBM. (1959). *IBM 1401 programming systems* [Brochure]. https://www.computerhistory.org/brochures/doc-4372957097531/

International Organization for Standardization. (2015). *Systems and software engineering—Framework for categorization of IT systems and software, and guide for applying it* (ISO/IEC TR 12182:2015). https://www.iso.org/standard/63611.html

Kotliarskyi, A., Zhu, V., & Brock, Z. (2026, April 27). *An open-source spec for Codex orchestration: Symphony*. OpenAI. https://openai.com/index/open-source-codex-orchestration-symphony/

Martin, J. (1991). *Rapid application development*. Macmillan Publishing Company. https://books.google.com/books?id=o6FQAAAAMAAJ

Northrop, L. M., Clements, P. C., Bachmann, F., Bergey, J. K., Chastek, G., Cohen, S. G., Donohoe, P., Jones, L. G., Krut, R. W., Jr., Little, R., McGregor, J., & O'Brien, L. (2012). *A framework for software product line practice, version 5.0*. Software Engineering Institute, Carnegie Mellon University. https://www.sei.cmu.edu/library/a-framework-for-software-product-line-practice-version-50/

OpenAI. (2025, May 16). *Introducing Codex*. https://openai.com/index/introducing-codex/

Qian, C., Liu, W., Liu, H., Chen, N., Dang, Y., Li, J., Yang, C., Chen, W., Su, Y., Cong, X., Xu, J., Li, D., Liu, Z., & Sun, M. (2024). ChatDev: Communicative agents for software development. In L.-W. Ku, A. Martins, & V. Srikumar (Eds.), *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)* (pp. 15174–15186). Association for Computational Linguistics. https://doi.org/10.18653/v1/2024.acl-long.810

Ruby on Rails. (n.d.). *The Rails command line*. https://guides.rubyonrails.org/command_line.html

van der Aalst, W. M. P., ter Hofstede, A. H. M., Kiepuszewski, B., & Barros, A. P. (2003). Workflow patterns. *Distributed and Parallel Databases*, *14*(1), 5–51. https://doi.org/10.1023/A:1022883727209

Zloof, M. M., & de Jong, S. P. (1977). The system for business automation (SBA): Programming language. *Communications of the ACM*, *20*(6), 385–396. https://doi.org/10.1145/359605.359615