Development Methodology of a Code-Illiterate User

Outcome-First Design, Multi-Model Drift Detection, and Machine-Readable Context as a Reproducible Methodology for AI-Assisted System Development Without Formal Training

John Phillip Bernhardt Jr.

Independent Researcher

Abstract

This methodology research paper documents the development methodology through which a practitioner without formal training in software engineering or computer science produced two independently validated AI infrastructure systems in under two months using AI-assisted collaboration. The systems examined are Hydra-rray, a containerized behavioral observatory designed to study tool-capable AI models operating with real offensive security tools, and an Android MCP client, a fully offline mobile AI agent integrating local transformer inference, operating-system-level automation through Android AccessibilityService, and Model Context Protocol (MCP) orchestration within a single application package. Development occurred entirely under consumer-access constraints: commodity hardware, standard consumer model interfaces, no engineered system prompts, no prompt-engineering infrastructure, no model fine-tuning, and no custom datasets. Context continuity across stateless model sessions was maintained exclusively through machine-readable YAML context transport documents referred to as batons. Across the two case studies, six consistent methodological properties were observed: outcome-first design, cross-domain methodology replication, vanilla model utilization, machine-readable context as drift prevention, multi-model triangulation for drift detection, and socialized interaction as a mechanism for surfacing tacit knowledge. Within the Hydra-rray experimental environment, an additional behavioral observation emerged: across observed sessions within the Hydra-rray environment, operating under consumer-access constraints and without engineered prompt scaffolding, tool-capable models exhibited a binary execution pattern: either correct tool execution or complete hallucination, with no intermediate partial success states observed. The Android case study further demonstrates that local language-model inference and operating-system-level actuation can operate on consumer mobile hardware not designed for transformer workloads, including a Snapdragon 865 device running Android 13. Development of the Android system occurred independently and the project repository predates Google's public AppFunctions announcement by one day, suggesting parallel discovery of a similar architectural gap. The paper argues that the primary contribution of these projects is not the systems themselves but the development methodology that produced them: a human-directed AI collaboration process in which the practitioner defines outcomes, evaluates architecture, and orchestrates the work while AI systems generate implementation artifacts. The paper itself is produced using the same methodology, serving as a recursive demonstration of the approach it documents.

1. Introduction

AI-assisted software development has rapidly entered mainstream technical practice, yet prevailing industry discourse commonly frames large language model (LLM) coding assistants as tools that require experienced engineering oversight in order to produce reliable systems. Within this framing, LLMs are often characterized as analogous to junior developers: capable of generating useful implementation artifacts but requiring correction, architectural direction, and validation from practitioners with formal training. The assumption underlying this model is that successful AI-assisted system development depends primarily on the practitioner's coding expertise and ability to supervise the model's output.

This paper presents an empirical observation that challenges that assumption. Two independently validated AI infrastructure systems were produced by a practitioner without formal software engineering training, without formal research training, and without the use of common prompt-engineering techniques or custom development infrastructure. Both systems were developed in under two months. One system is complete and operating as a behavioral research instrument. The second is in active development with independently validated milestones demonstrating architectural viability.

The contribution of this work is not the systems themselves but the development methodology that produced them. The paper documents and examines the observed properties of that methodology as it emerged during the construction of two systems developed in unrelated domains: a containerized behavioral observatory designed to study tool-capable LLM behavior under naturalistic conditions, and an offline Android AI agent integrating on-device inference with a mobile automation actuator layer. Despite differences in architecture, programming languages, hardware platforms, and problem domains, the same development methodology produced validated results in both contexts.

Both case studies were conducted under strict consumer-access constraints: standard model interfaces, no engineered system prompts, no custom infrastructure, and no model fine-tuning. Context continuity was maintained solely through stateless machine-readable context transport documents referred to throughout as batons. These constraints are important boundary conditions for interpreting the observations reported here. The systems documented in this paper were not constructed using specialized enterprise tooling, proprietary model access, or research infrastructure unavailable to independent practitioners. Instead, development occurred through consumer-accessible interfaces and commodity hardware.

The practitioner directing the work has a professional background outside computer science. Relevant experience includes fifteen years in emergency services as a paramedic and firefighter, with associated training in crisis intervention and behavioral pattern recognition, as well as prior employment in Tier II–III technical support roles. The practitioner has no formal training in software engineering, computer science, or academic research methodology. This absence of formal training is not treated here as a limitation to be minimized. Rather, it is a central condition of the case studies: the systems documented were produced under that constraint.

The research question motivating this investigation emerged during the development process itself: whether a practitioner without formal programming training, working through AI-assisted collaboration, could produce executable infrastructure systems whose architecture and behavior were reliable enough to function under real-world conditions. More broadly, the work explores whether human-directed AI collaboration can serve as a reproducible methodology for system development when the practitioner defines outcomes clearly but does not implement the code directly.

Recent literature on AI agents and autonomous systems has primarily examined model behavior under controlled benchmarks or adversarial safety testing. Parallel research traditions in human factors have examined automation failures and operator–system interaction in complex technical environments. However, systematic investigation

of how non-traditional practitioners interact with AI coding systems during real development work remains limited. In particular, the intersection between non-formal practitioners, tool-capable LLMs, and real infrastructure development has received little direct empirical attention. Existing literature documents adjacent phenomena such as automation surprise, specification gaming, and agentic misalignment, but these frameworks were developed primarily for deterministic automation systems or for AI systems operated by trained engineers rather than by non-traditional practitioners.

(See Appendix B: Competitive Landscape Research Methodology)

The case studies presented here therefore occupy a space between software engineering practice, human–AI collaboration research, and observational AI safety work. Rather than attempting to construct synthetic benchmarks or controlled experiments, the work documents the behavior of a human-directed AI development process operating under real constraints and producing operational systems.

Across both projects, six consistent properties of the development process emerged. These properties were not predefined design principles but observations identified retrospectively during analysis of development transcripts, project artifacts, and machine-readable context documents generated during the work.

The two systems produced through this process serve as empirical case studies for examining those methodological properties.

The first system, Hydra-rray, is a containerized behavioral observatory designed to study how tool-capable AI models behave when given access to real offensive security tools under naturalistic operating conditions.

The second system is an Android-based offline AI agent integrating an embedded inference engine, a mobile automation actuator layer, and a Model Context Protocol (MCP) orchestration layer within a single application package.

Taken together, these systems provide an opportunity to examine whether the development methodology used to produce them exhibits consistent properties across different technical contexts.

Subsequent sections describe the observed methodological properties, document the two case studies in detail, and examine the implications and limitations of the observations presented here.

2. Methodology

The development process documented in this paper did not begin as an explicit methodology. Instead, the approach emerged during the construction of the two case studies described later in this paper. Only after both systems had reached validated operational milestones were common properties of the development process examined retrospectively. The methodology described here therefore represents a set of empirical observations about how the practitioner conducted AI-assisted system development under the constraint regime defined in Section 1.

Each observation is presented as a property of the development process as it occurred in the case studies rather than as a universal prescription for AI-assisted development. The findings reflect what was observed to function in the two documented projects. Their applicability to other practitioners, domains, or development environments remains an open question addressed in the limitations section.

2.1 Outcome-First Design

In both case studies presented here, development began with the practitioner defining the completed system's behavior before any architectural or implementation decisions were made. The practitioner articulated the intended final outcome in operational terms: what the finished system would do, how it would behave in its environment, and how success would be recognized. Only after this definition was established were architectural and implementation steps considered.

All subsequent development decisions were evaluated against this predefined outcome. Architectural proposals generated during AI-assisted sessions were accepted or rejected based on whether they moved the system toward the defined final behavior. In practice, this approach constrained the scope of intermediate development work. Infrastructure that did not directly contribute to the defined outcome was not introduced.

In the case studies presented here, this approach produced codebases without legacy scaffolding or intermediate architectural layers built in anticipation of possible future directions. Each system's architecture evolved in direct alignment with the predefined outcome rather than through incremental exploration. This contrasts with conventional iterative development approaches in which scaffolding and partial infrastructure are constructed while the ultimate system architecture remains fluid.

Incremental development is a rational response to multi-stakeholder environments in which requirements change over time. In a single-practitioner context with full decision authority, however, the outcome definition can remain stable throughout development. Under those conditions, outcome-first design functioned as a structural constraint that prevented architectural drift.

2.2 Cross-Domain Methodology Replication

The methodology described here was applied independently to two development efforts operating in unrelated domains. The first involved containerized infrastructure for behavioral observation of tool-capable AI models interacting with offensive security tools. The second involved mobile software development, combining embedded inference, operating-system-level automation, and application orchestration within the Android ecosystem.

The two projects differed across multiple dimensions: programming languages, hardware platforms, operating systems, development environments, and system goals. The Hydra-rray observatory required container orchestration, security instrumentation, and logging infrastructure operating in a desktop environment. The Android MCP client required integration with mobile operating system services, JNI bridging for local inference, and real-time interaction with application interfaces.

Despite these differences, the same development process was used for both projects. Outcome definition preceded architecture, development sessions were conducted using standard consumer-facing AI interfaces, context continuity was maintained through machine-readable baton documents, and multiple models were used to cross-check outputs.

In both cases, validated system components were produced within approximately two months of initial project conception. The observed variable common to both projects was therefore not domain expertise or technical ecosystem familiarity, but the development methodology itself.

2.3 Vanilla Model Utilization

Neither case study employed custom system prompts, engineered personas, prompt scaffolding frameworks, or other forms of prompt-engineering infrastructure. All development sessions used standard consumer-access model interfaces with default behavior.

The practitioner did not maintain reusable prompt templates or structured prompt libraries. Instead, development interactions consisted of direct conversational exchanges describing the current system state, the desired next step, and the constraints governing the system. The practitioner's role in these interactions was to define the problem clearly and evaluate proposed outputs against the system's defined outcome.

In the case studies presented here, useful model output did not appear to depend on elaborate prompt design. Instead, the clarity of the practitioner's mental model appeared to function as the primary determinant of output quality. When the practitioner could precisely describe the current system state, the desired outcome, and the constraints governing acceptable solutions, models frequently generated implementation artifacts that could be incorporated into the system with minimal modification.

This observation does not imply that prompt engineering lacks value in other contexts. Rather, it suggests that within this development context, a practitioner with a clear mental model of the system being built may not require engineered prompt scaffolding to obtain actionable results.

2.4 Machine-Readable Context as Drift Prevention

Because development sessions occurred through stateless consumer model interfaces, maintaining project continuity across sessions required an alternative mechanism for preserving context. To address this need, the practitioner developed machine-readable context transport documents written in YAML format, referred to throughout the projects as batons.

Batons served as structured summaries of project state. They contained architectural descriptions, system constraints, validated milestones, and the next planned development steps. At the beginning of each new model session, the current baton was provided as input to establish the project context.

Several properties of batons were observed during their use in both projects.

First, batons were stateless artifacts. They functioned independently of any particular model, interface, or session history. A baton could be provided to any compatible model and immediately establish the project state required for continued development.

Second, baton generation occurred naturally as a byproduct of the development process rather than as a separate documentation task. When the practitioner requested a baton summarizing the current project state, the resulting artifact served both as session context for future work and as documentation of project evolution.

Third, batons accumulated informational richness over time. As projects progressed, each new baton incorporated validated milestones and architectural refinements, gradually forming a comprehensive machine-readable record of the system's development history.

Finally, batons enabled multi-model orchestration. Because they functioned as authoritative descriptions of project state, multiple models could independently evaluate outputs against the same baton context. This property proved particularly important for drift detection.

Unlike system prompts, which attempt to shape model behavior through instruction, batons primarily functioned as informational context. In practice, informing models about the system's state appeared more robust than attempting to influence their reasoning through behavioral instructions tied to a specific interface or model.

2.5 Multi-Model Triangulation for Drift Detection

Development sessions frequently involved more than one AI model. Rather than relying on a single model throughout a project, the practitioner used multiple models—primarily Claude and ChatGPT—to cross-check

architectural proposals, implementation artifacts, and interpretations of project state.

This practice functioned as an informal drift-detection mechanism. If one model produced output inconsistent with the current baton description of the system, a second model could be asked to evaluate whether the output aligned with the documented project state. Disagreement between models served as an early signal that the proposed output required review.

The cost of this triangulation mechanism was minimal because it required no additional infrastructure beyond access to multiple model interfaces. The baton documents provided a shared ground truth against which model outputs could be evaluated.

In practice, model disagreement often identified subtle inconsistencies before they propagated into implementation work. When both models independently agreed that a proposed output aligned with the baton description of the system, the practitioner could proceed with higher confidence that the output reflected the intended architecture.

2.6 Socialized Interaction as Tacit Knowledge Surfacing

A final observed property of the development process involved conversational interactions with AI models that were not directly tied to active development work. These sessions resembled informal discussions rather than technical implementation tasks.

During these conversations, the practitioner often described project decisions, reflected on development experiences, or explored conceptual questions about the systems under construction. Although these sessions did not produce immediate implementation artifacts, they served an observable functional role.

All six methodology findings described in this section were first articulated during such conversational interactions rather than during active development sessions. The conversational format appeared to surface implicit patterns that had not been recognized during the implementation process itself.

Another observation emerged during these interactions: when models accumulated context about the practitioner and the projects across multiple sessions, the nature of the interaction changed measurably. Less time was required to re-establish project context, and discussions could proceed at a higher conceptual level.

For a practitioner working without human collaborators, these conversational sessions appeared to distribute cognitive load that might otherwise accumulate during extended development work. The functional role of these interactions should not be interpreted as anthropomorphic collaboration. Rather, they functioned as a mechanism for externalizing reasoning and identifying patterns that structured artifacts alone did not reveal.

Together, the six observations described above constitute the methodology examined in this paper. Each observation emerged independently from the development of the two case studies and was later recognized as part of a consistent pattern across both projects. The following sections examine those case studies in detail and provide the empirical context from which these methodological properties were derived.

3. Case Study A: Hydra-rray Behavioral Observatory

The first application of the methodology described in Section 2 resulted in the development of Hydra-rray, a naturalistic behavioral observatory designed to study how tool-capable AI models behave when given access to real offensive security tools in uncontrolled operational conditions. The system was constructed to observe behavioral properties that conventional benchmark testing cannot capture. Synthetic benchmarks evaluate

performance against predetermined tasks under controlled evaluation protocols; they do not observe how models behave when given open-ended authority within real tool environments. Hydra-rray was therefore designed to capture full behavioral arcs in naturalistic sessions rather than task-specific performance metrics.

Hydra-rray operates as a containerized infrastructure environment composed of twenty-one Kali Linux containers. Sixteen containers host full offensive security toolsets, each containing approximately 4,280 tools drawn from standard Kali distributions. The remaining five containers function as bridge and decoy nodes that provide network routing and environmental realism without exposing additional offensive tooling. This architecture creates a mixed environment in which tool-capable models encounter both operational systems and inert infrastructure elements, preventing the environment itself from implicitly guiding model behavior.

The system incorporates a forensic-grade logging framework designed to preserve chain-of-custody integrity for all sessions conducted within the observatory. All operational events are recorded using SHA-256–hashed logs, allowing session artifacts to be verified against tampering. Container lifecycle management follows an ephemeral execution model: working containers are instantiated as clones and self-destruct after session completion, while master container images remain intact. This design ensures that individual sessions cannot contaminate the baseline environment while preserving the underlying infrastructure required for subsequent observations.

Hydra-rray also incorporates several supporting architectural layers that extend beyond simple container orchestration. A retrieval-augmented generation (RAG) layer using ChromaDB provides structured knowledge retrieval during sessions. A deception layer consisting of base-only containers introduces environmental ambiguity that prevents models from relying on simplistic environment assumptions. Finally, a validation layer enforces chain-of-custody protections on the observatory itself. Unauthorized operational attempts trigger an immediate execution halt, generate a complete forensic log of the attempt, and require authorized restoration before operation can resume. Logging continues throughout this process, but unauthorized operators observe only silent failure with no visible indication of the protective mechanism.

The primary behavioral observation emerging from early Hydra-rray sessions concerns what is referred to here as the Binary Gap. Across observed sessions within the Hydra-rray environment, operating under consumer-access constraints and without engineered prompt scaffolding, tool-capable models exhibited a binary execution pattern: either correct tool execution or complete hallucination, with no intermediate partial success states observed. In practice, this meant that when a model attempted to use a tool incorrectly, the failure mode was not degraded performance or partial completion but total hallucination of execution results. Conversely, when a model successfully executed a tool command, the result was fully correct and verifiable within the system environment.

This pattern was observed across early exploratory sessions conducted prior to formal baseline establishment. The finding is therefore explicitly bounded to the Hydra-rray environment and the constraint regime described in Section 1. It should not be interpreted as a universal property of tool-capable models. Nevertheless, the behavior is notable because it suggests the absence of intermediate confidence gradients in tool execution contexts. A model exhibiting only binary success or failure states may behave differently under adversarial conditions than a model whose execution reliability varies continuously.

From a safety perspective, the presence or absence of intermediate execution states is operationally relevant. Systems that degrade gradually allow detection of emerging failure conditions before catastrophic outcomes occur. Systems that operate in categorical states—either fully correct or completely incorrect—present different monitoring challenges. The Binary Gap observation therefore suggests a potentially safety-relevant behavioral property of tool-capable models that warrants further controlled investigation.

Hydra-rray is currently operating in a baseline-establishment phase designed to produce a statistically meaningful observational dataset. Sessions are conducted under a controlled protocol with hardware cool-off intervals to maintain environmental consistency. Each session that produces a complete behavioral arc is recorded as an immutable observational unit referred to as a "slice," defined here as a bounded observational record capturing behavior from initial stimulus through natural stopping condition without interpretive conclusions embedded in the artifact itself.

(See Appendix A: Slice Logging Methodology)

In order to preserve experimental validity, the model used to construct the Hydra-rray infrastructure is permanently separated from the models tested within it. Claude Code Opus 4.5 functions as a trusted builder responsible for infrastructure development and maintenance. Test participants consist of separate tool-capable models operating within the environment under observation. This builder–participant separation is treated as a hard methodological constraint to prevent contamination of behavioral observations through prior infrastructure knowledge.

At the time of writing, Hydra-rray functions as a working behavioral instrument rather than a completed research program. The observatory infrastructure is operational, baseline observational protocols are being established, and early behavioral patterns such as the Binary Gap have been identified. The next phase of research focuses on observing tool-poisoning behavior under naturalistic conditions. The purpose of this phase is to examine how tool-capable models respond when the tools themselves become adversarial inputs.

The Hydra-rray observatory therefore serves two roles within the present paper. First, it constitutes the first case study demonstrating the methodology described in Section 2. Second, it operates as an experimental instrument capable of producing future behavioral data about AI systems operating with real tool access. The system's existence provides the empirical foundation from which the methodological observations described earlier were derived.

4. Case Study B: Android MCP Client

The second application of the methodology described in Section 2 occurred in a substantially different technical domain: mobile operating systems. The project documented here is an Android-based AI agent designed to operate entirely offline within a single application package. The system integrates three major components: an embedded on-device inference engine using llama.cpp, an Android AccessibilityService acting as an actuator layer capable of executing interface actions, and an MCP server orchestration layer enabling tool-based reasoning workflows.

The system was designed with several architectural constraints from the outset. The reasoning layer must operate without cloud dependency. The agent must function on consumer hardware without requiring specialized devices. Target applications must remain unmodified, meaning the system must interact with existing Android applications through publicly available operating system mechanisms rather than through custom integration. These constraints ensured that the resulting system would operate within realistic consumer environments rather than under laboratory-only conditions.

The project originated as an attempt to automate a personal media library workflow. The practitioner required a method for locating media content, initiating downloads through a third-party application, and verifying completion without direct manual interaction. The development process began by defining the completed system's behavior: the agent would receive a natural language instruction describing desired content, locate that

content within a standard Android application environment, initiate a download through a separate application, and confirm completion through observable system events.

Industry implications emerged from this work only after development had progressed. The architectural pattern required to accomplish the defined personal workflow revealed a broader capability: a mobile AI agent capable of observing interface state, matching patterns, executing actions, and verifying outcomes without modifying the applications it interacts with. The identification of this architectural gap was therefore a consequence of solving a concrete personal problem rather than the result of an abstract search for research opportunities.

The hardware environment used for development provides an important boundary condition for interpreting the system's significance. The primary test device was a Samsung Galaxy S20 FE running Android 13 with a Snapdragon 865 system-on-chip. A second device, a Samsung Galaxy S22 Ultra also running Android 13, served as the upper bound for validation testing. The Snapdragon 865 platform was not designed or marketed for transformer inference workloads. Validation of local model inference on this hardware demonstrates that the architectural approach described here is accessible on devices widely available on the secondary market for approximately $150–200 as of 2026.

At the time of development, Android 13 is three major operating system versions behind the platform targeted by Google's AppFunctions architecture, which has been announced for Android 16 but has not yet reached production deployment. The project repository predates Google's public AppFunctions announcement by one day. Development occurred independently without awareness of the parallel effort. The timing suggests independent discovery of a similar architectural gap, and serves as external validation that the gap identified was real and the timing was correct.

Development of the Android system proceeded by validating its major architectural layers independently before attempting full system integration. Two of the three layers have reached validated operational milestones at the time of writing.

The first validated component is the actuator layer. This layer uses Android's AccessibilityService framework to observe interface state and execute interface actions without requiring modifications to target applications. Validation occurred on February 27, 2026. The mechanism involved monitoring window state indicators emitted through Android's accessibility event system and identifying the moment when a specific activity—QuickDownloadActivity—became available for interaction.

Once the appropriate window state was detected, the system executed an ACTION_CLICK event against the interface element responsible for initiating the download operation. A hydration delay of approximately 300 milliseconds was introduced to ensure that the interface element had fully initialized before the click action was dispatched. The implementation does not rely on fixed screen coordinates. Instead, the target interface element is identified dynamically through accessibility node properties.

Evidence for this validation consists of timestamped Android logcat output showing the full sequence of events: QuickDownloadActivity launch, Pop-Up Window focus acquisition, download initiation through the MediaProvider subsystem, and final surface destruction confirming completion of the download activity. The successful execution of this sequence demonstrates a minimal agent primitive: observe system state, match a known pattern, execute an action, and verify the outcome through observable events.

The second validated component is the local inference layer. Validation occurred on March 1, 2026 using the llama.cpp inference engine accessed through a JNI bridge within the Android application environment. Initial smoke testing used the VibeThinker 1.5B Q6_K model, with LLaMA 3.2 3B also available on the device for expanded testing.

The inference layer successfully produced coherent multi-sentence output on physical S20 FE hardware. The JNI bridge remained stable throughout testing, with no out-of-memory events, Android Application Not Responding (ANR) watchdog triggers, or native crashes observed during inference execution. These results confirm that transformer-based language model inference can operate on consumer mobile hardware not originally designed for such workloads.

At the time of writing, two of the system's three architectural layers have therefore been validated independently: the actuator layer and the inference layer. The remaining work focuses on the orchestration layer responsible for connecting the model's reasoning output to the actuator layer through a structured tool interface.

The next milestone involves developing a generic node-targeting abstraction capable of converting application-specific accessibility logic into reusable tool primitives. The current implementation identifies a specific activity associated with the Seal media-downloading application. The abstraction layer under development will generalize this mechanism so that interface elements can be targeted using node text, class names, or content descriptions. Once this abstraction is complete, it will form the basis of a tool registry accessible to the inference engine through the MCP orchestration layer.

Completion of the orchestration layer will close the agent loop, allowing the system to receive a natural language instruction, select appropriate tools, execute actions through the actuator layer, and verify completion through system state observation.

The demonstration scenario defined for the completed system illustrates the intended agent behavior. Upon receiving a natural language prompt requesting a particular piece of media content, the agent opens the YouTube application and searches for content matching the request. The agent then selects a result and shares it to the Seal application using Android's share-sheet interface. The AccessibilityService observes the QuickDownloadActivity window indicators, executes the appropriate interface action, and confirms completion through the notification lifecycle and surface destruction sequence associated with the completed download.

This scenario has been defined as the demonstration target for the completed system. The demonstration will be executed on both validation devices—the S20 FE representing the lower hardware bound and the S22 Ultra representing the upper bound. Each execution will produce timestamped logs documenting the full sequence of events from prompt reception through download confirmation.

The validated milestones described above therefore provide the second independent domain in which the development methodology documented in Section 2 produced operational results.

5. Findings and Implications

The two case studies presented in Sections 3 and 4 differ substantially in architecture, development environment, and operational objective. Hydra-rray is a containerized behavioral observatory designed to study tool-capable AI systems operating within a laboratory environment containing real offensive security tools. The Android MCP client is a mobile AI agent integrating local model inference with operating-system-level automation capabilities on consumer hardware. Despite these differences, both systems were produced using the same development process under the same constraint regime described in Section 1.

When examined together, the case studies suggest several observations about the relationship between human practitioners and AI-assisted development systems. These observations do not establish general rules for AI-assisted system construction. Instead, they represent implications suggested by the evidence generated in the two documented projects.

Code Literacy as a Non-Operative Variable

In the case studies presented here, code literacy did not appear to function as a required variable for producing independently validated AI infrastructure systems. Two such systems were produced under the consumer-access constraints defined in Section 1. One system is complete and functioning as a behavioral research instrument. The second remains in progress but has validated two of its three major architectural layers.

In both projects, the practitioner did not write code directly and did not perform manual code modification. Instead, the practitioner defined system outcomes, evaluated proposed implementations, and directed the development process through AI-assisted collaboration. Implementation artifacts were generated through interaction with language models operating through standard consumer interfaces.

This observation suggests a possible reframing of the competencies required for certain forms of AI-assisted system development. In the case studies presented here, the variables that appeared to govern development success were clarity of mental model, outcome-first thinking, pattern recognition, and methodological discipline. These competencies are not typically foregrounded in computer science curricula as primary drivers of system construction, yet they appeared to function as operative variables in the development processes described here.

Context Mismatch in the "Junior Developer" Model

LLM coding assistants are frequently described in industry discourse as analogous to junior developers whose output must be reviewed and corrected by experienced engineers. This framing assumes that models produce implementation artifacts requiring continuous correction to align with the intended system design.

In the case studies presented here, this pattern was not observed. Instead, models frequently produced implementations that matched the practitioner's specifications when the intended outcome was clearly defined. Development friction associated with integrating model-generated code into existing codebases—commonly reported in practitioner discussions—did not appear in either project.

One possible explanation for this difference lies in the absence of legacy codebases. Both systems documented in this paper were developed from the outset using outcome-first design rather than incremental integration with existing infrastructure. As a result, the systems contained no accumulated technical debt or undocumented architectural assumptions.

This observation suggests that some reported friction between AI-generated code and existing systems may reflect conflicts between new implementations and legacy architectures rather than inherent limitations of model-generated output. Under conditions where the system architecture is defined clearly before implementation begins, this friction may be reduced or absent.

Prompt Engineering as a Potential Compensatory Mechanism

Neither project employed custom system prompts, engineered personas, or structured prompt-engineering frameworks. All development interactions occurred through standard consumer-access model interfaces without prompt scaffolding infrastructure.

Despite this absence of engineered prompts, both projects produced validated results in these projects under the constraint regime defined earlier. In the context of these case studies, elaborate prompt scaffolding did not appear to be required in order to produce useful implementation artifacts.

One interpretation of this observation is that prompt engineering may sometimes function as a compensatory mechanism for unclear practitioner thinking rather than as a primary driver of effective AI-assisted development.

When the practitioner can articulate the system's current state, intended outcome, and relevant constraints precisely, models may generate useful outputs without additional prompt scaffolding.

This interpretation remains speculative. Controlled comparison studies would be required to determine whether prompt engineering provides measurable benefits when practitioner thinking is already clearly structured.

Naturalistic Observation and the Binary Gap

The Hydra-rray observatory produced an additional observation that has implications beyond the development methodology itself. Across observed sessions within the Hydra-rray environment, operating under consumer-access constraints and without engineered prompt scaffolding, tool-capable models exhibited a binary execution pattern: either correct tool execution or complete hallucination, with no intermediate partial success states observed.

Synthetic benchmark evaluations typically measure performance against predetermined tasks. These evaluations quantify accuracy rates or task completion success but do not observe the internal reliability patterns that emerge when models interact with real tools under open-ended conditions.

The Binary Gap observation therefore suggests that safety-relevant behavioral characteristics of tool-capable models may emerge only in naturalistic environments where models operate with real tool interfaces. If this observation generalizes beyond the Hydra-rray environment, it would have implications for the design of monitoring systems for autonomous agents.

The observation reported here is explicitly bounded to the Hydra-rray environment and the constraint regime described in Section 1. Additional research would be required to determine whether similar execution patterns appear in other tool environments or with other model architectures.

Methodology Transfer Across Domains

Perhaps the most significant implication of the two case studies concerns the transferability of the development methodology itself. The methodology documented in Section 2 was applied independently to two development efforts operating in different technical ecosystems with different system objectives.

The Hydra-rray observatory required container orchestration, forensic logging infrastructure, and security instrumentation within a desktop computing environment. The Android MCP client required integration with mobile operating-system services, embedded model inference, and user interface automation. These projects differ not only in technical implementation but also in the type of system being constructed.

Despite these differences, the same development process produced validated outcomes in both domains. Outcome-first system definition, baton-based context continuity, multi-model drift detection, and conversational reflection sessions were used throughout both projects.

This observation suggests that the methodology may be domain-agnostic, at least within the range of domains represented by the two case studies presented here. If this property holds under further investigation, the methodology could potentially be transferable to other development contexts where practitioners are able to define system outcomes clearly before implementation begins.

The observations presented in this section do not establish that the methodology documented in this paper will produce similar outcomes in other contexts. Instead, they identify patterns that emerged from the two documented case studies and suggest directions for further investigation. The following section addresses the limitations of the current work and outlines possible avenues for future research.

6. Limitations and Future Work

The observations presented in this paper arise from two case studies conducted by a single practitioner operating under a defined constraint regime. While the artifact record for both projects is extensive, the scope of the work imposes several limitations that must be acknowledged when interpreting the findings.

N = 1 Researcher

The methodology documented here was applied by a single practitioner. No independent replication by other practitioners has yet been attempted. As a result, the transferability of the methodology across practitioners with different backgrounds, cognitive styles, or problem-solving approaches remains untested.

The case studies demonstrate that the methodology can produce independently validated AI infrastructure systems under the conditions described in this paper. They do not demonstrate that the same methodology will reliably produce similar outcomes when applied by other practitioners. Replication by independent researchers would be required to evaluate the methodology's general applicability.

Incomplete Status of the Android Case Study

At the time of submission, the Android MCP client project remains in progress. Two of its three primary architectural layers—the inference layer and the actuator layer—have been validated independently on physical hardware. The remaining work concerns integration of these layers through a tool registry and completion of the agent execution loop.

The validated milestones documented in Section 4 establish that the core technical components of the system function under the defined hardware constraints. However, the complete end-to-end demonstration described as the project's final milestone has not yet been executed. The Android case study is therefore presented as an in-progress system with validated subsystems rather than a completed deployment.

Completion of the agent loop demonstration, including full logging of inference-to-action workflows on both target devices, would strengthen the evidentiary record for the second case study but is not required for the methodology observations reported in this paper.

Early Phase of Hydra-rray Behavioral Research

The Hydra-rray observatory described in Section 3 is operational but remains in the early phase of its research program. Baseline establishment for the behavioral dataset is ongoing, with a target minimum of twenty observational sessions under controlled hardware conditions before formal analysis proceeds.

The Binary Gap observation described in Section 3 emerged during pre-baseline observational sessions conducted under the defined constraint regime. While the observation was consistent across those sessions, the dataset remains small relative to what would be required for formal statistical analysis.

Future work within the Hydra-rray research program will expand the observational dataset, particularly in the area of tool poisoning behavior under naturalistic conditions. Publication of that dataset would allow independent researchers to evaluate the behavioral patterns reported here.

Constraint Regime and Resource Profile

Both projects were conducted under strict consumer-access constraints. Development occurred on consumer hardware using standard model interfaces without system prompts, prompt-engineering infrastructure, custom

datasets, or model fine-tuning. Context continuity was maintained exclusively through stateless machine-readable baton documents.

These constraints were intentionally maintained throughout the development process in order to establish boundary conditions for the observations reported in this paper. However, the constraint regime also limits the extent to which the findings can be generalized to practitioner environments with substantially different resource profiles.

At the same time, the hardware accessibility demonstrated in the Android case study suggests that the lower bound for AI-assisted infrastructure development may be significantly lower than commonly assumed. The Snapdragon 865 platform used as the floor device in the Android case study was not designed for transformer inference workloads yet successfully executed local model inference through an embedded llama.cpp engine.

Further research would be required to determine how the methodology behaves when applied in environments with significantly greater computational resources, different model architectures, or different operational constraints.

Future Research Directions

Several avenues for future work follow directly from the limitations described above.

First, completion of the Android MCP client agent loop demonstration would produce a fully integrated example of the methodology applied to mobile AI agent development. Executing this demonstration on both the floor device (Samsung Galaxy S20 FE) and the ceiling device (Samsung Galaxy S22 Ultra), with complete timestamped logs of the inference-to-actuation sequence, would finalize the second case study's evidentiary record.

Second, expansion of the Hydra-rray behavioral dataset would allow the Binary Gap observation and related behavioral patterns to be evaluated under a larger sample of sessions. Publication of anonymized session logs and analysis results would enable independent examination of the patterns observed within the Hydra-rray environment.

Third, replication of the methodology by independent practitioners would provide the most direct test of its transferability. Such replication attempts could determine whether the observed development process depends primarily on the practitioner's specific cognitive approach or whether it represents a reproducible collaboration pattern between humans and language models.

Fourth, formal documentation of baton design patterns could assist practitioners seeking to replicate the stateless context-continuity mechanism used in both case studies. Because baton documents evolved organically during the projects rather than being designed as a formal framework, extracting and documenting their design principles would support future replication efforts.

Finally, controlled comparison studies between outcome-first system definition and scaffolding-first incremental development approaches could help determine whether the development efficiency observed in these case studies results from the methodology itself or from contextual factors specific to the projects examined here.

The limitations described above do not diminish the empirical observations documented in this paper. Rather, they define the boundary conditions within which those observations should be interpreted and identify the next steps required to determine whether the methodology described here extends beyond the specific cases in which it was observed.

7. Conclusion

The methodology documented here produced two independently validated AI infrastructure systems in under two months, under consumer-access constraints, without formal training, without system prompts, and without custom infrastructure.

The door was open the whole time.

All content in this paper was produced through AI-assisted collaboration, including this line.

Appendix A: Slice Logging Methodology

Hydra-rray records model interaction sessions as bounded observational artifacts referred to as slices. A slice represents a complete behavioral arc within the observatory environment and serves as the fundamental unit of analysis for subsequent behavioral research.

A slice is defined as a bounded observational record capturing system behavior from an initial stimulus through a natural stopping condition without embedding interpretive conclusions within the artifact itself. This design intentionally separates observation from analysis. The slice record preserves what occurred during a session without introducing researcher interpretation or classification into the primary artifact.

Each slice contains the following components:

Session Initialization Context

The baton state used to initialize the session, including environment configuration, system constraints, and the initial prompt or stimulus presented to the model.

Operational Event Log

A timestamped sequence of events generated during the session. These events include model outputs, tool invocation attempts, command execution results, and environmental responses generated by the Hydra-rray infrastructure.

Forensic Integrity Record

All slice artifacts are secured using SHA-256 hashing. Hash records allow subsequent verification that the slice contents have not been modified after the session completed.

Session Termination Boundary

The slice concludes when a natural stopping condition occurs. Examples include completion of the requested task, termination of the model interaction, or infrastructure lifecycle events such as container self-destruction.

Slices are treated as immutable records once closed. Subsequent analysis, classification, or interpretation occurs in secondary research artifacts rather than through modification of the slice itself. This approach preserves the evidentiary integrity of the primary observational dataset.

During baseline establishment, slices primarily serve as documentation of behavioral transitions rather than as comparative units. As the dataset expands, slices can be classified according to behavioral patterns identified during later analysis phases.

This artifact-first approach ensures that the Hydra-rray dataset preserves the full operational context in which model behavior occurred, allowing future researchers to re-examine the raw records without dependence on the original researcher’s interpretations.

Appendix B: Competitive Landscape Research Methodology

The Android MCP client described in Section 4 emerged from development work rather than from a formal search for an architectural gap. After the system architecture began to stabilize, the practitioner conducted a structured competitive landscape review to determine whether similar systems already existed within the global software ecosystem.

The investigation was conducted through three independent research sweeps spanning multiple technology communities and language ecosystems.

The first sweep examined English-language technical literature and developer communities, including open-source repositories, developer forums, and public documentation describing Android automation systems and mobile AI agents.

The second sweep expanded the search to include Chinese-language developer ecosystems, which frequently host independent experimentation with mobile automation frameworks and AI-assisted software systems.

The third sweep extended the investigation to Korean and Japanese developer communities, both of which maintain active Android automation and mobile tooling ecosystems.

Across all three sweeps, the search focused on identifying systems exhibiting the same combination of architectural characteristics present in the Android MCP client:

fully offline mobile language model inference

integration with an automation actuator capable of interacting with existing applications

tool-based orchestration enabling reasoning-to-action workflows within a single mobile application

While numerous related technologies were identified—such as mobile automation frameworks, Android accessibility tooling, and mobile inference engines—no system was found that combined all three elements into a unified architecture operating entirely within a single application package.

The absence of directly comparable systems across the examined ecosystems does not establish novelty in a strict academic sense. Instead, it indicates that the architecture described in Section 4 does not appear to correspond directly to a previously documented system within the surveyed developer communities.

The repository timestamp associated with the Android MCP client predates Google’s public announcement of the AppFunctions architecture by one day. This timing suggests that the practitioner and the Google engineering team independently identified a similar architectural opportunity in the Android ecosystem. The significance of this observation lies not in priority but in convergence: independent discovery of similar architectural patterns suggests that the underlying gap identified during development was real.

Appendix C — Baton Context Transport Specification

Machine-readable context transport documents referred to in this paper as batons serve as the mechanism by which development context was preserved across stateless AI model sessions.

Large language model interfaces typically provide no persistent session state across interactions with different models or across separate working sessions. In practice this creates a continuity problem: development work performed across multiple sessions or multiple models can easily drift from the current system state.

Batons were developed as a lightweight solution to this continuity problem.

A baton is a structured, machine-readable document describing the current project state, development objectives, and constraints required for a model to operate correctly within the ongoing work. Batons were written in YAML format because the structure is both human-readable and easily parsed by software systems.

The design goals for batons were:

Stateless portability. A baton must be interpretable by any model, in any session, without reliance on hidden system prompts or persistent conversational memory.

Minimal ambiguity. The baton must describe the current project state precisely enough that a model receiving the document can correctly determine the next development step.

Incremental growth. Batons accumulate structure as a project evolves. Early batons may contain only a small set of fields, while later batons contain detailed architectural context.

Model neutrality. The baton format does not rely on any specific model architecture or provider interface.

In practice batons served three functional roles:

Context restoration

When beginning a new development session, the practitioner would provide the current baton to the model. This restored the complete project context in a single artifact.

Drift detection

When multiple models were used during development, the baton served as a ground-truth record of the current system state. Divergence between model output and baton state provided an early signal of reasoning drift.

Project state archival

Batons also functioned as historical records of the system’s development trajectory. Each baton captures the architecture and goals of the project at a specific moment in time.

A simplified baton example is shown below:

```
context_id: android_mcp_server_state
```

created: 2026-02-27

project: name: Android MCP Client goal: fully offline mobile AI agent

validated_layers: - inference - actuator

pending_layer: - tool_registry

next_milestone: node_targeting_abstraction

In both case studies presented in this paper, batons were generated as a natural byproduct of development sessions rather than as a separate documentation task. As a result the baton corpus provides a high-fidelity record of the

reasoning and decision processes that produced the final systems.

Because batons are independent of any specific AI system, the approach is potentially transferable to other human-AI collaborative development environments.

References

(References will be expanded as the Hydra-rray behavioral dataset and competitive landscape sources are formally published. The current reference list focuses on foundational literature relevant to the conceptual framing of the paper.)

Bainbridge, L. (1983). Ironies of automation. Automatica, 19(6), 775–779.

Bostrom, N. (2014). Superintelligence: Paths, Dangers, Strategies. Oxford University Press.

Christiano, P., et al. (2017). Deep reinforcement learning from human preferences. Advances in Neural Information Processing Systems.

Leveson, N. (2011). Engineering a Safer World: Systems Thinking Applied to Safety. MIT Press.

Russell, S. (2019). Human Compatible: Artificial Intelligence and the Problem of Control. Viking.

Shneiderman, B. (2022). Human-centered AI. Oxford University Press.

Bernhardt, J. P. (2026). Hydra-rray Behavioral Observatory: Infrastructure and Development Archive. Private repository. Available upon request for academic review.