

PromptBridge: AI Coding Agents you can Trigger from Chat and Trust

Tripti Singh¹; Dolly Verma²; Parineeta Jha³

¹M.Tech Student, Department of CSE, RSR Rungta College of Engineering and Technology (C.G.)

^{2,3}Assistant Professor, Department of CSE, RSR Rungta College of Engineering and Technology (C.G.)

Publication Date: 2026/10/05

Abstract: AI coding agents such as Claude Code, Cursor, Codex, and OpenCode have matured into capable multi-step software engineers, yet nearly all of them still assume a developer is sitting at a terminal. PromptBridge removes that assumption by connecting six messaging platforms, Telegram, Discord, Slack, Microsoft Teams, GitHub, and Email, to four coding agents through a single dispatcher. This paper presents the complete, matured architecture of PromptBridge: the four-layer design, the session and git-aware change tracking model, the security and configuration layer, and the packaged Electron desktop application. Beyond describing the current system, this paper evaluates it against related agent frameworks and bot toolkits, sets out a formal threat model, examines the structural checkpoints that keep a human in the approval loop for every agent-initiated change, and lays out a concrete future scope covering sandboxed execution, richer output formatting, session summarization, and controlled user studies. The goal is to give a single, self-contained account of what PromptBridge is today and where it is going next.

Keywords: AI Coding Agents, Multi-Platform Orchestration, Session Management, Git-Aware Automation, Conversational Software Engineering, Security, Future Scope.

How to Cite: Tripti Singh; Dolly Verma; Parineeta Jha (2026) PromptBridge: AI Coding Agents you can Trigger from Chat and Trust. *International Journal of Innovative Science and Research Technology*, 11(9), 2895-2903.
<https://doi.org/10.38124/ijisrt/26sep1386>

I. INTRODUCTION

Agentic coding tools have advanced quickly. Claude Code, Cursor, Codex, and OpenCode can now read an entire codebase, plan a multi-file change, edit the files, run tests, and report the outcome with only a short natural-language instruction from the developer [1], [4]. SWE-bench [6] and SWE-agent [7] show that this is not a narrow demo capability; these agents resolve real, previously unseen GitHub issues in production-scale repositories.

Despite this progress, almost every one of these tools is built around the assumption that a developer is at a workstation with a terminal open. Software teams do not work that way all the time. A bug report lands in Slack, a review comment appears on GitHub, a client writes an email, and a manager asks a question in Teams. Each of these moments is a legitimate trigger for a coding task, but none of them happens at a terminal.

PromptBridge was built to close this gap by treating the messaging platform itself as the interface to the coding agent. Earlier work on this system examined its layered architecture and its session and change-tracking mechanisms separately. This paper consolidates both strands into one account of the system as it exists today, at version 2.0.0, and extends the discussion with a security analysis and a concrete roadmap

for future development, which prior work only touched on briefly.

The system now supports six messaging platforms and four coding agents through a shared dispatcher, keeps per-chat session state so multi-turn tasks do not need to be re-explained, reports exactly which files an agent changed after every run, and ships as a self-contained desktop application for Windows, macOS, and Linux that needs no public server or cloud account.

The contribution of this paper is not a new mechanism but a complete, current picture of a system that has grown across several development cycles, together with an honest accounting of its limits and a specific plan for what should be built next. This combination, current architecture plus forward roadmap, is what distinguishes this paper from earlier, narrower treatments of PromptBridge.

The rest of the paper is organized as follows. Section II reviews related work. Section III describes the research methodology used to build and evaluate the system. Section IV presents the current system architecture in full. Section V covers implementation details. Section VI evaluates extensibility, cost, and example workflows, including a formal threat model. Section VII discusses the design.

Section VIII addresses responsible AI and human oversight. Section IX sets out the future scope. Section X concludes.

The contributions of this paper are:

- A single, current description of the full PromptBridge architecture spanning platform adapters, the dispatcher, session state, git-aware change tracking, and the security and configuration layer.
- A design-and-build methodology for evolving a multi-platform integration layer without breaking existing adapters.
- A comparative evaluation against agent frameworks and bot toolkits, including extensibility metrics measured in lines of code.
- A concrete, prioritized future scope covering execution isolation, output formatting, context management, and empirical validation with real teams.
- A formal threat model of the system's assets and mitigations, and an analysis of the structural checkpoints that keep a human in the approval path for every agent-initiated file change.

II. RELATED WORK

➤ *Coding Agents and Benchmarks*

Codex [1] and GitHub Copilot [2] established that language models can generate working code from natural language prompts inside an editor. AlphaCode [3] extended this to competitive programming problems. SWE-bench [6] moved evaluation from isolated functions to full repositories, and SWE-agent [7] showed that the interface an agent uses to navigate a file system matters as much as the underlying model. Claude Code [4], Cursor [5], and Devin [8] are the practical descendants of this line of work, but all of them still expect the developer to be at a terminal or inside an IDE.

CodeGen [21] and DeepSeek-Coder [22] broadened the set of base models available for coding tasks, and the general trend across this body of work is that model capability has grown faster than the ways a developer can reach that capability day to day. PromptBridge is agnostic to which of these models sits underneath the agent; it treats Claude Code, Cursor, Codex, and OpenCode as interchangeable back ends behind a common runner interface, so improvements to any one of them reach the developer without any change to the delivery layer.

➤ *Agent Orchestration Frameworks*

ReAct [9] interleaved reasoning with tool calls, and AutoGen [10] generalized this to teams of cooperating agents. LangChain [11] and LlamaIndex [12] give developers building blocks for connecting models to data sources and tools. None of these frameworks specify how a task should arrive from a chat platform or how the result should be delivered back into that same conversation; that integration layer is left to the application built on top of them, which is exactly the layer PromptBridge provides.

MemGPT [23] is closer in spirit, treating limited context as a resource to be managed rather than ignored by

tiering older conversation into external storage. PromptBridge's session model currently keeps full history without this tiering, which is why session summarization in Section IX is framed as an adaptation of MemGPT's approach rather than a new idea.

➤ *Bot and Conversational Frameworks*

Dialogflow [13] and Rasa [14] handle natural-language dialogue for customer-facing bots but have no concept of file systems, git state, or long-running processes. Platform SDKs such as Slack Bolt [15] and the Microsoft Bot Framework [16] solve message delivery for one platform at a time; supporting a second platform means duplicating the workflow logic. PromptBridge defines one interface, BotContext, that every adapter implements, so the workflow logic is written once regardless of platform count.

Rich-messaging primitives such as Slack Block Kit and Microsoft Adaptive Cards [26] show that platforms increasingly support structured, interactive replies beyond plain text. PromptBridge currently uses the lowest common denominator, markdown, so that every platform receives something usable, which is functional but leaves value on the table on platforms that could render a much richer reply; this trade-off is revisited directly in Section IX.

➤ *Developer Context and Coordination Research*

LaToza et al. [17] found that context switching is one of the costliest activities in a developer's day because rebuilding a mental model takes real time. Bird et al. [18] found that messaging platforms are already the primary coordination point for distributed teams, and work on CI/CD bots [19] showed developers are already comfortable receiving automated reports inside those same channels. These findings motivate delivering agent output through chat rather than a separate dashboard.

Weissgerber et al. [25] found that small, well-scoped patches are reviewed faster than large ones, supporting the Section IV-C choice to report a changed-file list rather than a full diff.

➤ *Gap and Positioning*

Liu et al. [20] surveyed over a hundred LLM-based software agents and named integration with existing developer workflows as an open problem. No system in the surveys above combines multi-platform delivery, multi-agent execution, session continuity, and git-aware change reporting in one self-hosted package that needs no public server. This paper documents the system built to fill that gap and, unlike prior treatments, gives equal weight to what is missing and what should be built next.

It is worth being precise about what PromptBridge is not claiming: a better underlying model, agent-computer interface, or dialogue manager than the systems cited above, each remains someone else's active research problem. What it claims is that the layer connecting a chat message to any of those systems, and the result back to that same chat, has been under-addressed relative to how often developers need it, and

that this layer can be built once and shared across every platform and agent rather than rebuilt per combination.

III. METHODOLOGY

PromptBridge was developed using an iterative, design-and-build methodology rather than a controlled experiment. Each development cycle added one platform adapter or one agent runner, verified that it worked end to end against a live account for that platform, and then checked that no existing adapter or the dispatcher required modification. This cycle was repeated six times for platforms and four times for agents, and the resulting code was kept as the primary evidence that the shared-core design holds under real growth rather than only in theory.

Four design goals were fixed at the outset and used as acceptance criteria for every change. First, platform neutrality: the dispatcher must never import a platform SDK. Second, zero public infrastructure: every integration must work from a personal access token, a polling loop, or a socket connection rather than a webhook that needs a public address. Third, configuration without code edits: every runtime setting must be changeable from the web dashboard. Fourth, auditability: every agent run must produce a visible record of which files it touched.

Evaluation in this paper follows a design-review methodology common in systems research: extensibility is measured by lines of code required to add a platform or agent, coverage is measured by walking through concrete workflows end to end on each platform, and comparison against related tools is done along functional dimensions rather than a benchmark score, since no existing benchmark measures multi-platform delivery. This is an appropriate methodology for a systems and tooling contribution, though Section IX identifies controlled user studies as the natural next step.

Security review followed a threat-modeling approach rather than penetration testing. For each component, credential storage, the dashboard API, and the agent runner, the question asked was who can trigger it and what it can access if triggered. This produced the allowlist and bearer-token controls described in Section IV-E and also produced the specific gaps listed as future work in Section IX.

Backward compatibility was treated as a hard constraint: any change requiring an edit to an existing adapter was reworked until it could be expressed purely as an addition. This is what keeps the lines-of-code figures in Section VI-A meaningful, they measure isolated growth, not growth that hides coupling elsewhere in the codebase.

IV. SYSTEM ARCHITECTURE

PromptBridge is built around four layers plus a security and configuration layer that cuts across all of them. Fig. 1 shows the overall structure. A message enters through a platform adapter, is normalized into a BotContext, handed to the dispatcher, and the dispatcher hands agent execution to a

runner; results flow back through the same path in reverse. No layer skips ahead of the next one, which keeps the control flow easy to reason about even as the number of platforms and agents grows.

➤ Platform Adapter Layer

Six adapters, Telegram, Discord, Slack, Teams, GitHub, and Email, each translate their native event format into a shared BotContext object exposing sendMarkdown, sendText, editMessage, showTyping, sendFile, sendWithButtons, acknowledgeAction, and updateButtonMessage. The GitHub adapter polls issue and pull request comments every 120 seconds for /claude, /cursor, /codex, or /opencode, which avoids any webhook or public endpoint. The Email adapter runs an IMAP IDLE loop with exponential backoff and buffers all outbound messages into one consolidated reply per run.

➤ Dispatcher and Session Layer

The dispatcher is the single place where workflow logic lives; it never imports a platform SDK. For every incoming event it resolves the active project and session for that chat, selects the agent, manages the run, and sends results back through the same BotContext. Session state is a small record per chat and project pair, project id, agent, model, and history, loaded fresh on every message rather than kept in memory, so a restart never loses conversation context. This is what lets a developer say "now check if that broke anything" and have it mean something.

Keying sessions by the pair of chat identifier and project identifier, rather than by chat alone, lets one developer hold independent conversations about two different projects inside a single channel, and lets a shared team channel hold one session per project regardless of which team member sent the last message. The dispatcher also exposes /project, /model, and /session as first-class commands so this state can be inspected and reset without editing any file on disk.

➤ Runner Layer and Git-Aware Change Tracking

Each agent, Claude Code, Cursor, Codex, and OpenCode, has a runner that builds the correct command-line invocation, streams output back to the dispatcher as progress, and reports changed files when the process exits. Before the agent runs, the runner snapshots the project: the current git status for a git repository, or a timestamp otherwise. Afterward, it compares the new git status against that snapshot, or looks for files modified since that timestamp. The result is a short list of file paths, not a diff, sent alongside the main output, giving the developer a review starting point without flooding a chat thread.

➤ File Attachment Pipeline

Inbound files, a screenshot, a log, a specification document, are downloaded to a local directory and referenced in an enriched prompt so the agent can read them as ordinary project files. Outbound files created or modified by the agent are uploaded back through the same channel. This closes a gap the session model alone does not: a task can begin with a document rather than only with text.

Email attachments follow the same pattern through a dedicated inbox folder, and inline images used purely for message formatting are skipped so only files the developer actually intended to share reach the agent. A per-attachment size limit prevents a single oversized file from stalling a run before the agent even starts.

➤ *State, Configuration, and Security Layer*

Two JSON files, projects and sessions, hold all persistent state, and a third, settings, holds platform tokens,

allowlists, and agent paths, all editable from the dashboard without touching a file by hand. Access to any platform is gated by a user allowlist; the dashboard REST API is gated by a bearer token, with only /api/status left public. Configuration is cached with a short time-to-live so dashboard edits, including a token rotation that should restart a running adapter, take effect within seconds. This is an appropriate control set for personal and small-team use; Section IX lists what a larger shared deployment would still need.

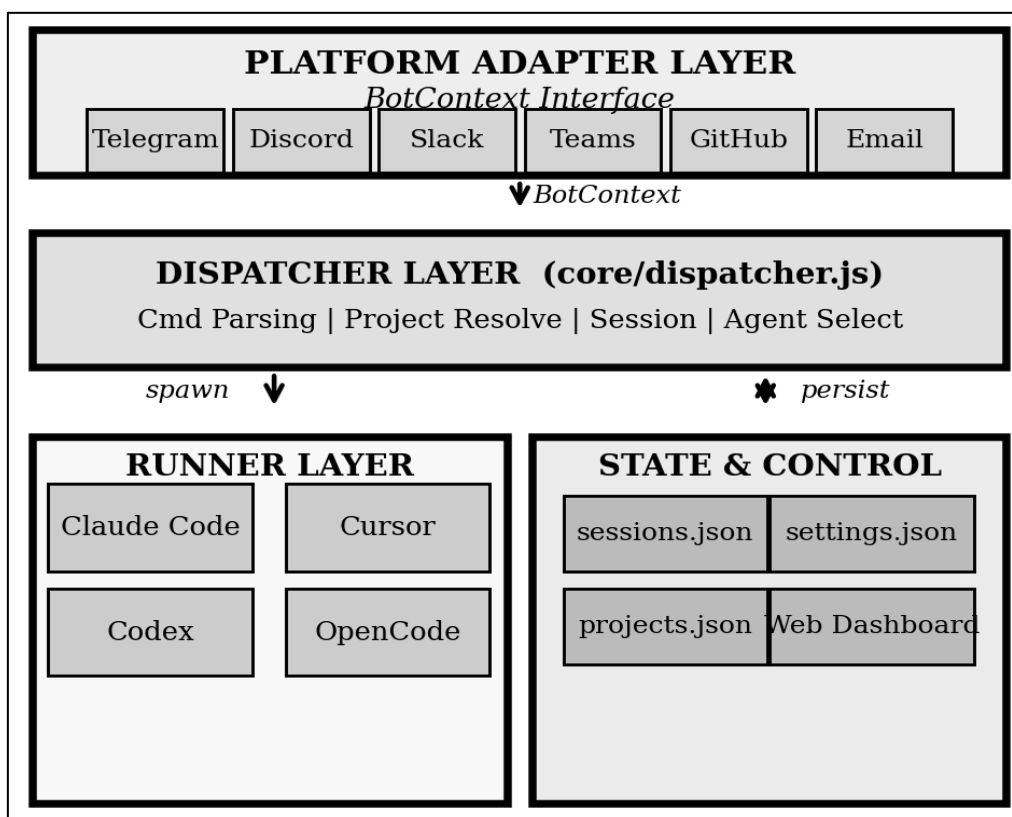


Fig 1 PromptBridge Four-Layer Architecture with the Cross-Cutting Security and Configuration Layer.

V. IMPLEMENTATION

PromptBridge is implemented in Node.js 18 or later. Every adapter lives in its own directory and imports only the BotContext interface from the shared core, never another adapter. Configuration loads from environment variables for bootstrap-only values, the dashboard port and its bearer token, and from the settings file for everything else, so runtime behavior never depends on redeploying the process.

Concurrency is tracked per conversation. If a user sends a second command while an earlier run on the same project is

still active, the dispatcher declines the new command with a short explanation rather than starting a conflicting run, which would otherwise corrupt the git change report and possibly the working tree itself.

The REST API exposes project, session, and settings management, plus platform lifecycle control; Table 1 lists the primary endpoints. /api/run/email lets an external system, a CI pipeline or a scheduled job, trigger a run and receive the result by email without going through any chat platform at all.

Table 1 Primary REST API Endpoints

Method	Path	Purpose
GET	/api/status	Health, version (public)
GET/PUT	/api/settings	Read/update runtime config
POST	/api/run/email	Trigger run, email result
POST	/api/platforms/:n/start	Start a platform adapter
POST	/api/platforms/:n/stop	Stop a platform adapter

The packaged Electron application wraps the same server code, routes control through a context-bridge IPC channel rather than exposing Node APIs to the renderer, and stores all data in the OS user data folder so no manual path configuration is needed on Windows, macOS, or Linux.

Adding a new coding agent needs only a new runner function, roughly eighty lines, that builds the invocation, forwards output, and reports changed files; nothing else in the system changes. Adding a new platform needs only a new

adapter implementing BotContext, typically between 160 and 220 lines.

VI. RESULTS

The design described above runs as a working system rather than a prototype. Fig. 2 shows the web dashboard's project view, the primary interface an operator uses to add, edit, and monitor the projects PromptBridge can run agents against.

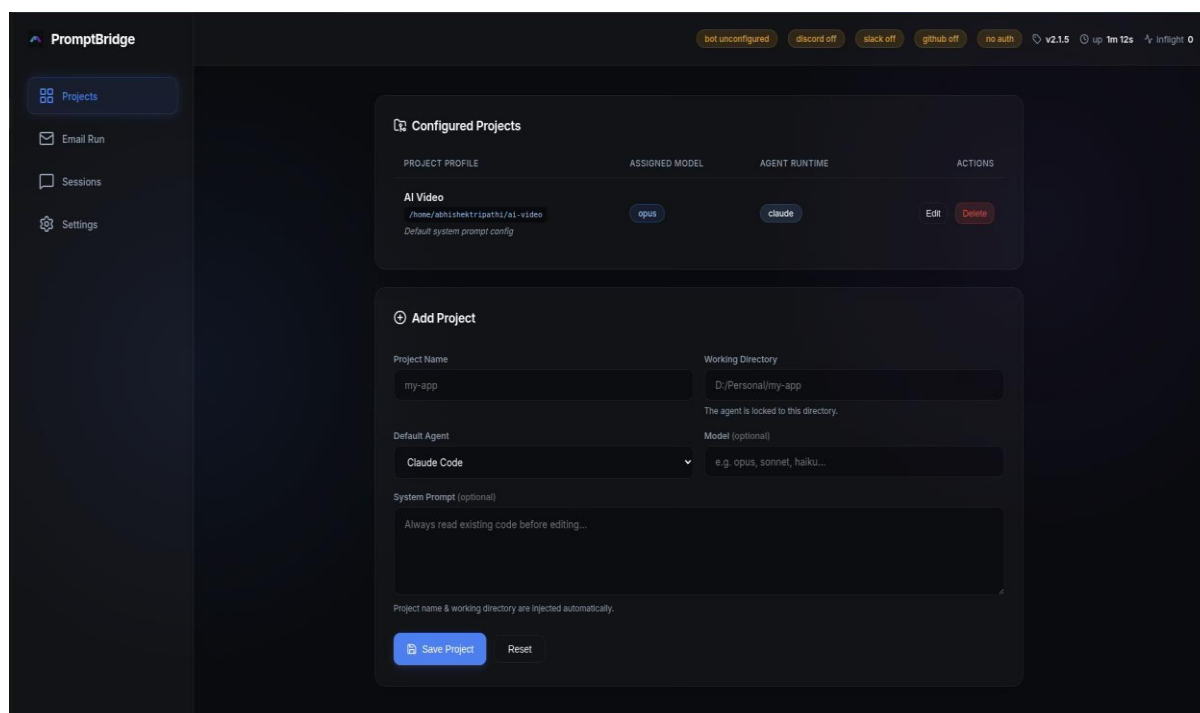


Fig 2 PromptBridge Dashboard, Project Management View.

Each row corresponds to a project record: its working directory, default agent, and default model, all editable without touching a configuration file by hand. This is the same dashboard referenced throughout Sections IV and V; showing it here ties those design claims to the interface an operator actually sees.

VII. EVALUATION

➤ Extensibility

The shared dispatcher is about 430 lines and handles every platform identically; each adapter is self-contained at 160 to 220 lines. Roughly two-thirds of the functional workflow logic is therefore shared rather than duplicated, so a dispatcher fix reaches all six platforms simultaneously without any platform-specific patch.

This ratio has held steady as platforms were added: the sixth adapter, Email, needed about the same number of lines as the first, Telegram, and required zero changes outside its own directory. That consistency, rather than any single adapter's line count, is the strongest evidence that the BotContext boundary is doing its job.

➤ Comparison with Related Tools

Table 2 compares PromptBridge with agent frameworks and bot toolkits on functional coverage. Frameworks like LangChain and AutoGen provide agent logic but no ready path from a chat platform to that logic; platform SDKs like Slack Bolt provide delivery but no agent execution or change reporting. PromptBridge is the only entry offering all of these together with no public server requirement.

Table 2 Comparison with Related Tools

Feature	PromptBridge	LangChain	AutoGen	Slack Bolt	Rasa
Coding agent integration	Yes	Partial	Partial	No	No
Multi-platform support	Yes (6)	No	No	No (1)	No
Session continuity	Yes	No	Partial	No	Yes
Git change reporting	Yes	No	No	No	No
No public server required	Yes	No	No	No	No

➤ Example Workflows

A developer on a phone sends /claude check the failing tests in Slack; PromptBridge resumes the stored session, runs Claude Code, and posts results with a changed-file list to the same thread. A reviewer comments /codex clean up the error handling on a GitHub pull request; the polling adapter picks it up within two minutes and replies on the same comment thread, no webhook required. A developer emails hi /claude summarize this week's changes; the IMAP listener matches the project from the subject line and replies with a single consolidated summary and any generated files attached.

➤ Response Time Characteristics

PromptBridge itself adds roughly one to two seconds of overhead per run for session loading and result formatting; total latency is dominated by the agent, seconds for simple questions to minutes for large refactors. The GitHub adapter

is the exception, bounded below by its 120-second poll interval, which is fine for asynchronous review threads but not for interactive waiting.

➤ Security Posture

The allowlist and bearer-token controls were checked against three questions: who can start a run, who can read or change settings, and who can reach the dashboard API at all. Every path answered correctly for a personal or single-team deployment, an unlisted user cannot trigger a run on any platform and no endpoint that changes state is reachable without the token, but none of the three paths currently distinguishes between two authorized users, which is the specific gap Section IX-2 targets. Table III sets out this analysis as a formal threat model: the assets the system holds, the corresponding threat to each, and the mitigation currently in place.

Table 3 Threat Model: Assets, Threats, and Mitigations

Asset	Threat	Current Mitigation
Platform tokens & credentials	Leaked via file or backup access	Local settings file only; never transmitted off-device
Dashboard REST API	Unauthorized settings or project changes	Bearer token required; only /api/status public
Project file system	Agent reads/writes unintended files	No sandbox yet; scoped to allowlisted users (Section IX-1)
Chat platform access	Unauthorized user triggers a run	Per-platform allowlist by user ID or username
Session history	Sensitive conversation content at rest	Local JSON in OS user data folder; not encrypted (future work)
Inbound email attachments	Oversized or malformed file stalls a run	25 MB per-file cap; inline images skipped

The table makes the biggest remaining gap explicit: project file system access is scoped by who can reach the system at all, not by what any one authorized user should be allowed to touch. That distinction is exactly what role-based access control, listed in Section IX-2, is meant to close.

VIII. LIMITATIONS

The system does not sandbox agent execution, so an agent can touch any file inside its configured project directory. Session history has no automatic size limit, so a very long-running conversation can eventually push the agent past its context window. Teams and Email have limited markdown support, so headings and code blocks sometimes render as plain text on those two channels only.

The evaluation itself has a limitation worth stating directly: it was carried out by the system's own developers rather than an independent team, and Table II is a functional checklist rather than a quantitative benchmark, since no shared benchmark for multi-platform agent delivery yet exists. Section IX treats closing this gap as a priority rather than a footnote.

IX. DISCUSSION

The central design decision in PromptBridge is the BotContext interface. By pushing every platform-specific detail behind eight fixed methods, the dispatcher becomes

testable with a mock adapter and immune to the churn of six different third-party SDKs changing over time. Every improvement made to the dispatcher benefits all six platforms with no additional work.

Git-aware change reporting turned out to matter more than expected. Knowing exactly which files an agent touched, without reading a full diff in a chat thread, gives a developer a concrete starting point for review and is the single feature most responsible for making remote, terminal-free agent runs feel trustworthy rather than opaque.

Choosing polling over webhooks for GitHub was a direct consequence of the zero-public-infrastructure goal. A webhook needs a reachable address, DNS, and usually a reverse proxy; a personal access token needs none of that, which is what makes the GitHub integration usable from an ordinary developer laptop on a home or office network.

The current security model, allowlists plus a bearer token, is appropriate for personal and small-team use but stops short of what a shared, multi-team deployment would need. This gap, rather than being treated as an afterthought, is carried forward directly into the future scope in Section IX as the highest-priority item.

Packaging the whole system as a single Electron desktop application turned out to matter for adoption as much as any protocol decision. A developer who would never set

up a systemd service or a Docker container will install and open a desktop application, and the control window's per-platform start and stop buttons make the state of the system visible in a way a headless server process is not.

X. RESPONSIBLE AI AND HUMAN OVERSIGHT

An agent that only answers questions is easy to trust: a wrong answer wastes time. An agent that edits, deletes, or creates files on a real project carries a different kind of risk, and that risk does not go away because the request arrived from a phone in Slack instead of a terminal. Parasuraman and Riley's classic finding that people both misuse automation, trusting it beyond what it has earned, and disuse it, ignoring it even when it is reliable [27], applies directly to a system whose whole purpose is to make triggering an agent effortless.

PromptBridge keeps three structural checkpoints between a message and an executed file change, rather than relying on the developer to remember to be careful. First, the allowlist gate in Section IV-A means no run starts at all unless the sender is already authorized for that platform. Second, the runner never commits or pushes on the developer's behalf; an agent's edits land in the working tree and stay there until the developer reviews and commits them through their own tooling. Third, the git-aware change report described in Section IV-C is sent on every run without exception, so there is always a visible record of what happened even if the developer chooses not to read it immediately.

The report format is itself a safeguard against automation bias: a full diff is easy to skim past on a phone screen, but a short file list is short enough that a developer is more likely to actually look at it. This trade-off is deliberate but incomplete, a file list says what changed, not whether the change is correct, so it reduces rather than removes the need for real review. Data handling stays local by the same logic: session history, settings, and attachments live only in the operator's own files, and what the underlying model provider does with a prompt once the runner hands it off is outside this system's control and a dependency an operator should weigh before connecting sensitive code.

None of this amounts to a guarantee. It is a set of checkpoints that keep a human in the approval path for every file change without adding a second step for the developer to remember, which is the only kind of safeguard likely to survive contact with a busy team. Section IX-2 extends this from checkpoints to a full audit trail, and Section IX-1 extends it from an allowlist to a sandbox; both harden this same oversight model rather than replace it.

XI. FUTURE SCOPE

The evaluation and discussion above point to specific, concrete extensions rather than open-ended improvement. These are listed here in the order they would most change the system's reach and trustworthiness.

- Sandboxed execution: run each agent inside a container or restricted process, using a technology such as gVisor [24], so file and command access is limited to what a task actually needs. This directly addresses the main limitation noted in Section VI-F and is the change most needed before any shared, multi-user deployment.
- Role-based access and audit logging: move beyond a flat allowlist to per-project permissions, so a user can be authorized for one project without gaining access to all of them, plus a persistent, queryable log of every command and file change for accountability in multi-team deployments.
- Session summarization: automatically compress older turns once a conversation approaches the agent's context window, following the tiered-memory approach of MemGPT [23], so long-running sessions do not silently lose early instructions the developer still expects the agent to remember.
- Native output formatting: convert markdown to Adaptive Cards for Teams and to HTML for Email, using the same rich-messaging primitives referenced in Section II-C [26], so headings, lists, and code blocks render correctly on every platform, not just the ones that support markdown natively.
- Controlled user studies: measure task completion time, error rate, and developer satisfaction with real teams under realistic conditions, replacing the current design-review evaluation with the empirical evidence identified as missing in Section VI-F.
- Additional platforms and a plugin model: extend beyond the current six platforms, WhatsApp and Matrix are natural candidates given their large developer communities, through a documented adapter interface that external contributors can implement without touching the dispatcher or any existing adapter.
- Multi-agent collaboration: allow a single task to be split across more than one agent run concurrently, for example one agent per module in a large refactor, with the dispatcher merging the individual change reports into one combined summary.
- CI/CD-triggered runs: let a pipeline failure automatically open a session and request an agent fix, closing the loop between automated testing and automated remediation without waiting for a developer to notice the failure first.

None of these items require a change to the four-layer architecture itself; each is an addition within an existing layer, which is the strongest practical evidence that the architecture set out in Section IV has room to grow.

XII. CONCLUSION

PromptBridge connects six messaging platforms to four AI coding agents through a four-layer architecture, a shared dispatcher, per-chat session state, and git-aware change tracking, all governed by a lightweight security and configuration layer. This paper has given a single, current account of that full system at version 2.0.0, evaluated it against related agent and bot frameworks, and, unlike earlier

treatments, set out a specific and ordered future scope rather than a general statement of intent.

The evaluation shows that roughly two-thirds of the system's functional logic is shared rather than duplicated across platforms, that a new agent can be added in about eighty lines of code, and that the system is the only one among its comparators to offer coding agent execution, multi-platform delivery, and git-aware change reporting together without any public server requirement.

The future scope identified here, sandboxed execution, role-based access, session summarization, native output formatting, controlled user studies, additional platforms, multi-agent collaboration, and CI/CD-triggered runs, is deliberately scoped to extend the existing architecture rather than replace it, which is itself evidence that the architectural choices made early in the project have held up as the system has grown.

PromptBridge's value has always been in the connection it provides, between the messaging platforms developers already use and the coding agents that are becoming capable enough to act on their behalf, and this paper has aimed to leave a clear, complete, and forward-looking record of that connection as it stands today.

As agents keep improving, the bottleneck in practice shifts from what an agent can do to how easily a developer can ask it to do that thing at the moment the need arises. PromptBridge is an argument, made in working code across six platforms and four agents, that closing that second gap is worth as much engineering attention as closing the first, and the future scope in Section IX is written to keep that argument true as both the agents and the platforms around them continue to change.

REFERENCES

- [1]. M. Chen et al., "Evaluating large language models trained on code," arXiv preprint arXiv:2107.03374, 2021, doi: 10.48550/arXiv.2107.03374.
- [2]. A. Ziegler, E. Kalliamvakou, X. A. Li, A. Rice, D. Rifkin, S. Simister, G. Sittampalam, and E. Aftandilian, "Productivity assessment of neural code completion," in Proc. 6th ACM SIGPLAN Int. Symp. Mach. Program. (MAPS), 2022, pp. 21–29, doi: 10.1145/3520312.3534864.
- [3]. Y. Li et al., "Competition-level code generation with AlphaCode," *Science*, vol. 378, no. 6624, pp. 1092–1097, 2022, doi: 10.1126/science.abq1158.
- [4]. Anthropic, "Claude Code," [Online]. Available: <https://code.claude.com/docs/en/overview>. Accessed: Sep. 21, 2026.
- [5]. Anysphere, "Cursor: The AI Code Editor," [Online]. Available: <https://www.cursor.com/>. Accessed: Sep. 21, 2026.
- [6]. C. E. Jimenez et al., "SWE-bench: Can language models resolve real-world GitHub issues?," arXiv preprint arXiv:2310.06770, 2023, doi: 10.48550/arXiv.2310.06770.
- [7]. J. Yang, C. E. Jimenez, A. Wettig, K. Lieret, S. Yao, K. Narasimhan, and O. Press, "SWE-agent: Agent-computer interfaces enable automated software engineering," arXiv preprint arXiv:2405.15793, 2024, doi: 10.48550/arXiv.2405.15793.
- [8]. Cognition, "Introducing Devin, the first AI software engineer," [Online]. Available: <https://cognition.com/blog/introducing-devin>. Accessed: Sep. 21, 2026.
- [9]. S. Yao et al., "ReAct: Synergizing reasoning and acting in language models," in Proc. Int. Conf. Learn. Representations (ICLR), 2023.
- [10]. Q. Wu et al., "AutoGen: Enabling next-gen LLM applications via multi-agent conversation," arXiv preprint arXiv:2308.08155, 2023, doi: 10.48550/arXiv.2308.08155.
- [11]. H. Chase, "LangChain," GitHub repository, 2022. [Online]. Available: <https://github.com/langchain-ai/langchain>. Accessed: Sep. 21, 2026.
- [12]. J. Liu, "LlamaIndex," GitHub repository, 2022. [Online]. Available: https://github.com/run-llama/llama_index. Accessed: Sep. 21, 2026.
- [13]. Google Cloud, "Dialogflow," [Online]. Available: <https://cloud.google.com/dialogflow>. Accessed: Sep. 21, 2026.
- [14]. T. Bocklisch, J. Faulkner, N. Pawlowski, and A. Nichol, "Rasa: Open source language understanding and dialogue management," arXiv preprint arXiv:1712.05181, 2017, doi: 10.48550/arXiv.1712.05181.
- [15]. Slack Technologies, "Bolt for JavaScript," Slack Developer Documentation, [Online]. Available: <https://docs.slack.dev/tools/bolt-js/>. Accessed: Sep. 21, 2026.
- [16]. Microsoft, "Bot Framework SDK," GitHub repository, [Online]. Available: <https://github.com/microsoft/botframework-sdk>. Accessed: Sep. 21, 2026.
- [17]. T. D. LaToza, G. Venolia, and R. DeLine, "Maintaining mental models: A study of developer work habits," in Proc. 28th Int. Conf. Software Eng. (ICSE), 2006, pp. 492–501, doi: 10.1145/1134285.1134355.
- [18]. C. Bird, N. Nagappan, P. T. Devanbu, H. C. Gall, and B. Murphy, "Does distributed development affect software quality? An empirical case study of Windows Vista," in Proc. 31st Int. Conf. Software Eng. (ICSE), 2009, pp. 518–528, doi: 10.1109/ICSE.2009.5070550.
- [19]. D. Hilton, N. Nelson, T. T. D. Le, E. K. K. et al., "Usage, costs, and benefits of continuous integration in open-source projects," in Proc. 31st IEEE/ACM Int. Conf. Automated Software Eng. (ASE), 2016, pp. 426–437, doi: 10.1145/2970276.2970358.
- [20]. J. Liu, K. Wang, Y. Chen, X. Peng, Z. Chen, L. Zhang, and Y. Lou, "Large language model-based agents for software engineering: A survey," arXiv preprint arXiv:2409.02977, 2024, doi: 10.48550/arXiv.2409.02977.
- [21]. E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong, "CodeGen: An open large language model for code with multi-turn

- program synthesis,” arXiv preprint arXiv:2203.13474, 2022, doi: 10.48550/arXiv.2203.13474.
- [22]. D. Guo et al., “DeepSeek-Coder: When the large language model meets programming,” arXiv preprint arXiv:2401.14196, 2024, doi: 10.48550/arXiv.2401.14196.
- [23]. C. Packer, S. Wooders, K. Lin, V. Fang, S. G. Patil, I. Stoica, and J. E. Gonzalez, “MemGPT: Towards LLMs as operating systems,” arXiv preprint arXiv:2310.08560, 2023, doi: 10.48550/arXiv.2310.08560.
- [24]. S. Kim et al., “gVisor: Container security through kernel API sandboxing,” Google, Tech. Rep., 2019.
- [25]. P. Weißgerber, D. Neu, and S. Diehl, “Small patches get in!,” in Proc. 2008 Int. Working Conf. Mining Software Repositories (MSR), 2008, pp. 67–76, doi: 10.1145/1370750.1370767.
- [26]. Slack Technologies, “Block Kit,” Slack Developer Documentation, [Online]. Available: <https://docs.slack.dev/block-kit/>. Accessed: Sep. 21, 2026.
- [27]. R. Parasuraman and V. Riley, “Humans and automation: Use, misuse, disuse, abuse,” Human Factors, vol. 39, no. 2, pp. 230–253, 1997, doi: 10.1518/001872097778543886.