PromptBridge: A Multi-Platform Orchestration Framework for Remote AI Coding Agents

Tripti Singh[1], Dolly Verma[2], Nidhi Sharma[3]
[1] *M.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.)*

***Abstract*—AI coding agents like Claude Code and Cursor can now read files, edit code, and handle multi-step programming tasks. Most of these tools run in a terminal or an IDE. In real teams, however, work often starts in Slack, Telegram, GitHub, or email. Switching tools every time a developer wants to run an agent takes time and breaks focus. This paper presents PromptBridge, a lightweight framework that connects six messaging platforms to four AI coding agents through one shared workflow. When a developer sends a task on any supported platform, the system finds the right project, runs the right agent, and sends the result back to the same channel. The system tracks conversation history, handles file attachments, and reports which files changed after each run. A design review shows that the four-layer architecture makes it easy to add new platforms or agents without changing the shared core.**

***Index Terms*—AI coding agents, multi-platform integration, software engineering, orchestration, remote development, conversational interfaces**

I. INTRODUCTION

AI coding agents have become much more capable in the last two years. Tools like Claude Code, Cursor, and Codex can now read through a codebase, edit files across multiple directories, and carry out multi-step tasks with very little guidance from the developer. These tools are changing what it means to write software [4, 6]. Benchmarks like SWE-bench [6] show that modern agents can resolve real GitHub issues automatically. Devin [8] demonstrated that an agent can set up a repository, write tests, and fix failing builds without any human guidance at each step.

Despite this progress, most agents still require a developer to sit at a workstation with access to a terminal. This is a real barrier for modern software teams. Developers work across different time zones and locations. Some team members only have a phone or tablet when they need to act on a task. Others work in environments where opening a terminal is not possible at the moment the task arrives, such as during a meeting, on a commute, or when working from a device that does not have the agent installed.

In most software teams, work does not begin at the terminal. A bug report arrives through Slack. A code review note appears on GitHub. A client sends a request by email. A product manager posts a question in a Teams channel. The developer who receives this message must then switch tools, open the terminal, navigate to the right project directory, and type the correct command before the agent can even begin working. Each of these steps takes time and breaks the developer's focus on the original task.

Research on developer work habits shows that context switching is one of the most expensive activities in a software workflow [17]. Every time a developer leaves one tool to open another, they spend time rebuilding the mental context they need to work effectively. For remote teams and mobile workers, this cost is even higher because the terminal may not be immediately accessible at all. The result is that useful agents go unused or underused simply because reaching them is too much effort at the moment.

PromptBridge fills this gap. It sits between the messaging platform and the coding agent. When a developer sends a task in any supported channel, PromptBridge picks it up, runs the right agent on the right project, and sends the result back to the same place. The developer never has to leave the conversation. They can review the output, see which files changed, and decide what to do next, all from their phone, tablet, or desktop messaging app, without opening a terminal.

The system supports six messaging platforms: Telegram, Discord, Slack, Microsoft Teams, GitHub, and Email. It connects to four AI coding agents: Claude Code, Cursor, Codex, and OpenCode. A single dispatcher component handles all workflow logic. Platform-specific code stays behind a clean interface called BotContext. Adding a new platform or agent does not require any changes to the dispatcher or to any existing adapters.

AI coding agents are powerful but only useful when a developer can reach them. PromptBridge makes that reach as short as one message in any channel the developer is already watching. This removes the largest practical barrier to adopting these tools in everyday team workflows, not just for planned coding sessions at a workstation but for quick checks and small tasks that arise throughout the day in any location.

The rest of the paper is organized as follows. Section II reviews related work on coding agents, agent frameworks, bot frameworks, and developer tooling. Section III describes the methodology and design goals. Section IV covers the four-layer system architecture. Section V describes implementation details. Section VI evaluates extensibility and presents example workflows. Section VII discusses implications and limitations. Section VIII concludes the paper.

The contributions of this paper are:

1) A four-layer design connecting six messaging platforms to four coding agents through a single shared dispatcher.
2) A BotContext interface that isolates platform-specific code from workflow logic.
3) Session tracking and git-based change reporting across all supported platforms.
4) A design review showing that new platforms and agents can be added without changing the shared core.
5) A packaged desktop application for Windows, macOS, and Linux that runs the full system locally without a public server or cloud account.

II. RELATED WORK

A. AI Coding Agents and Benchmarks

The path from simple code completion to full software engineering agents spans about five years. Codex [1] was among the first models trained specifically on code, showing that language models could complete functions reliably from natural language descriptions. GitHub Copilot [2] turned this research into a widely used IDE tool and demonstrated measurable productivity gains in controlled studies. AlphaCode [3] pushed further and reached competitive programming performance on standard benchmark tasks.

SWE-bench [6] created a more realistic evaluation standard. It collects real GitHub issues and asks agents to fix them by modifying actual codebases. This benchmark revealed that completing isolated functions in a vacuum is very different from resolving real bugs in large repositories with many interacting files. SWE-agent [7] improved performance on SWE-bench by designing better agent-computer interfaces, showing that how an agent navigates the file system is as important as the quality of the underlying language model. CodeGen [21] and DeepSeek-Coder [22] extended the space of available base models for coding tasks.

Claude Code [4] and Cursor [5] represent the current generation of practical tools. Both work inside a developer's actual codebase and support multi-step file editing, test running, and shell command execution. OpenCode and Codex extend this with different model backends and interaction styles. All of these tools assume the developer is at a terminal, which is the problem that PromptBridge solves.

B. Agent Frameworks

ReAct [9] combined reasoning and acting in a single loop, giving language models a way to use tools and then reflect on the results before deciding the next action. This made multi-step problem-solving more reliable than earlier approaches that committed to a plan upfront without checking intermediate outcomes. AutoGen [10] extended this idea to multi-agent settings where specialized agents collaborate on complex tasks by passing structured messages back and forth.

LangChain [11] and LlamaIndex [12] gave developers a toolkit for building pipelines that connect language models to external data sources and tools. These frameworks are strong at composing logic between components and at chaining model calls. They do not address how a coding task reaches the framework from a messaging platform or how results get delivered back to a user in a channel they are already reading.

PromptBridge targets precisely this delivery and integration layer.

C. Bot Frameworks and Messaging

Dialogflow [13] and Rasa [14] are widely used for building conversational agents in customer service settings. They handle natural language understanding and manage dialogue across multiple conversation turns. Neither framework has built-in support for coding tasks, agent execution, or file system access. Their output is always text, not code changes or file diffs.

The Slack Bolt SDK [15] and Microsoft Bot Framework [16] each provide platform-specific libraries for building bots. These libraries are not interoperable. A bot built with Slack Bolt does not work on Teams, and a Teams bot does not work on Slack. If a team uses both platforms, all workflow logic must be written twice. PromptBridge avoids this by defining a single common interface that every platform adapter implements, so all workflow logic is written once in the dispatcher and runs on any supported platform.

D. Developer Tools and Remote Work

Research by LaToza et al. [17] showed that context switching is one of the most costly activities for software developers. When a developer switches tools to complete a task, they lose the mental context they were holding and must rebuild it when they return. This adds time and introduces errors. Bird et al. [18] studied distributed software teams and found that messaging platforms are the primary coordination point, making them a natural and low-friction place to trigger developer tools and receive their output.

Work on CI/CD bots [19] showed that developers are already comfortable receiving automated build reports and test results in messaging channels. They act on those reports without switching to a separate CI dashboard. This established pattern shows that delivering coding agent output through the same messaging channel would fit naturally into existing team workflows without requiring any new habits.

E. Surveys on LLM-Based Agents

Liu et al. [20] review over 100 papers on LLM-based software agents, covering code generation, bug fixing, test writing, and repository-level multi-step editing. The survey shows the field has moved from function-level tools to full repository-level agents that can navigate large codebases, run tests, and interpret failure messages. The authors identify the integration of agents with existing developer workflows as an open challenge. PromptBridge addresses this challenge by providing the integration layer that connects agents to the platforms developers use every day.

Across all of these related lines of work, a common gap appears: no existing system combines multi-platform messaging delivery, coding agent execution, session tracking, and change reporting in a single self-hosted framework that works without a public server. PromptBridge is designed specifically to fill this gap by treating the messaging platform as the primary interface for the entire coding workflow, from issuing the initial command to reviewing the final result.

III. METHODOLOGY

PromptBridge follows a layered approach to connect multiple messaging platforms with AI coding agents through a unified workflow. The framework is designed to reduce context switching by allowing developers to send commands from any supported messaging platform and receive results in the same place, without leaving the conversation or opening a separate terminal window.

The workflow begins when a user sends a command from a supported platform such as Slack, Telegram, Discord, Microsoft Teams, GitHub, or Email. The corresponding platform adapter receives the message and converts it into a standard internal format called BotContext. This conversion step is critical because each platform delivers messages in a different structure. Telegram uses update objects with optional photo or document fields. Discord sends interaction payloads with attachments. Slack sends event JSON with a different field layout. Teams wrap everything in an Activity object. By enforcing the BotContext interface, the rest of the system never needs to know which platform is in use.

The standardized request is then forwarded to the dispatcher, which acts as the central controller of the framework. The dispatcher identifies the associated project by looking up the conversation's chat ID and the selected project ID in the session store. It then selects the requested AI coding agent based on the command or the session default, spawns the agent

process, streams progress updates back to the user during the run, and waits for the process to exit. When the agent finishes, the dispatcher reads the git state to find which files changed and sends a formatted summary back through the platform.

The framework maintains project settings and session state separately from execution logic. Project records store the working directory, the selected agent, and the default model. Session records store the conversation history for each chat so the agent can refer back to earlier messages when they are relevant to the current task. This clean separation means the dispatcher can handle any project on any platform using the same code path with no duplication.

Three design goals guided every architectural decision. First, adding a new platform or agent must not require changes to any existing component. This means new code must be entirely self-contained. Second, the system must work without a public server so that developers can run it on a laptop without configuring DNS records, firewall rules, or cloud infrastructure. Third, operators must be able to configure all settings from a web browser without ever editing a file on disk by hand.

The session management design is a key methodology decision. Each session is identified by the combination of the messaging platform's chat identifier and the PromptBridge project identifier. This pairing means a single developer can keep independent sessions for different projects in the same conversation, and the same project can have separate sessions in different channels or on different platforms. Sessions are loaded fresh for each incoming message rather than held in memory, so the system can be restarted between messages without losing any conversation context stored in the session file.

Security was considered at the design level. Access to any platform is gated by an allowlist of user IDs or usernames. Only users on the allowlist can send commands that trigger agent runs. The dashboard API is protected by a bearer token. Settings, including platform credentials, are stored in a local JSON file that never leaves the operator's machine. These controls are appropriate for personal and small-team use. Larger deployments may want per-project access restrictions and audit logging of all commands.

IV. SYSTEM ARCHITECTURE

PromptBridge is built around four layers. Figure 1 illustrates the complete workflow of PromptBridge.

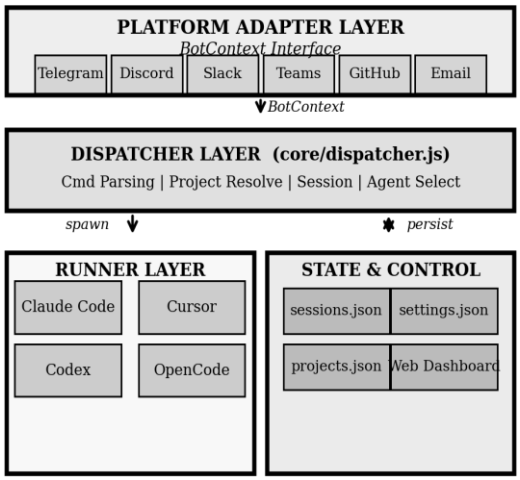

Fig. 1. PromptBridge four-layer system architecture.

A. Platform Adapter Layer

This layer handles all six platforms: Telegram, Discord, Slack, Teams, GitHub, and Email. Each platform has its own adapter that converts incoming messages, commands, and file attachments into a shared BotContext object. This object exposes a fixed set of methods: sendMarkdown sends formatted text, sendText sends plain text, editMessage updates a message already sent, showTyping sends a typing indicator, sendFile uploads a file, and sendWithButtons sends text with interactive buttons. The dispatcher only calls these methods and never imports any platform SDK directly. Fig. 2 shows the BotContext interface and how all six platform adapters implement it.

The GitHub adapter polls issues and pull request comments every 120 seconds looking for commands like /claude, /cursor, /codex, or /opencode. No webhook setup is needed because polling only requires read and write access on the repository through a personal access token. This makes the GitHub integration work on developer laptops and on internal networks where a public server address is not available. All other adapters receive events in real time using their respective SDKs.

The Email adapter handles both inbound and outbound directions. The IMAP listener runs an IDLE loop that wakes up when new messages arrive and uses

exponential backoff to reconnect automatically if the connection drops. Attachments from inbound emails are saved to a local inbox folder before the agent run begins. Outbound replies are buffered in memory during the run and sent as one consolidated email when the agent finishes, keeping the reply thread clean and easy to read.

B. Dispatcher Layer

The dispatcher is the core of the system. It receives a BotContext from an adapter, finds the active project and session for the conversation, selects the right agent, manages the run from start to finish, and sends progress and results back through the same BotContext. All workflow logic is in this one component. Because it only uses BotContext methods, the dispatcher code does not change when a new platform is added. Fig. 2 shows the execution flow through the dispatcher.

The dispatcher handles all supported commands. The /claude, /cursor, /codex, and /opencode commands start an agent run using the corresponding tool. The /project command shows a button menu for choosing the active project. The /model command shows a menu of available model presets. The /session command clears conversation history. The /help command lists all available commands. Button selections from earlier menus also flow through the dispatcher, which handles them as callback actions.

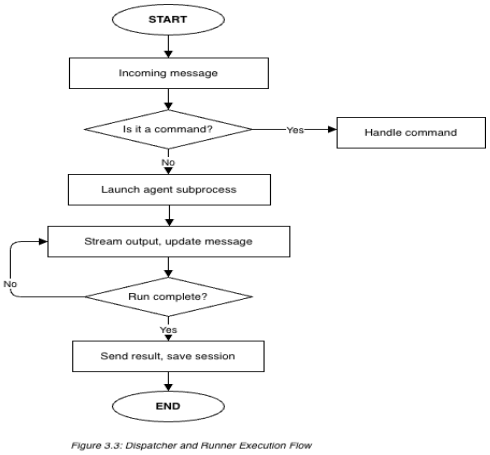

Figure 3.3: Dispatcher and Runner Execution Flow

Fig. 3. Dispatcher and runner execution flow.

C. Runner Layer

This layer runs the agent process. Each supported agent has a dedicated runner function that translates the internal request into the correct command-line invocation for that agent, streams output lines back to the dispatcher as they arrive, and returns a summary of changed files when the process exits. Runners read configuration fresh at the start of each run, so changing the agent path or model preset in the dashboard takes effect on the very next command without a restart.

The runner starts by recording the current git HEAD commit hash and a snapshot of file modification times for the project directory. It then spawns the agent as a child process with the project directory as the working directory and the session history as part of the input context. As the agent writes to its standard output, the runner forwards those lines to the dispatcher, which formats them and sends progress updates to the user at regular intervals so the channel shows activity rather than staying silent during a long run.

After the agent process exits, the runner checks the git working tree to see which files were modified, added, or deleted during the run. It compares the current git status against the recorded starting state and also checks which tracked files have a modification time later than the start of the run. The result is a short list of file paths that the dispatcher formats and sends to the user alongside the main agent output, giving the developer an immediate summary of what changed.

Error handling in the runner covers three cases. If the agent executable is not found at the configured path, the runner sends a clear error message describing the problem and the path that was checked. If the agent process exits with a non-zero status, the runner still collects and sends any output produced before the error, so the developer can see how far the agent got. If the run is interrupted because the user sends a stop command from the platform, the runner terminates the child process gracefully and reports the partial results.

D. State and Control Layer

Project settings and session data are stored in two JSON files. The settings file holds platform tokens, allowlists, agent executable paths, and SMTP configuration for Email. The sessions file maps each chat ID and project ID pair to a stored conversation history. A web dashboard lets the operator view and update all settings, add or remove projects, clear sessions, and start or stop individual platform adapters independently. Settings are cached with a short time-to-live so changes appear within a few seconds without requiring any manual refresh or process restart.

The web dashboard also serves as the primary interface for managing projects. Each project record contains the local path to the project directory, the default coding agent to use for that project, the default model preset, and an optional project name that appears in the selection menu. Operators can add, edit, and delete projects without restarting the system. When a project is deleted, its sessions are preserved separately so historical run data is not lost.

Platform adapter lifecycle is managed as a set of independent processes. Each adapter can be started or stopped through the dashboard or through the REST API without affecting any other running adapter. When an adapter is stopped, any active run on that platform continues to completion and the result is delivered normally. When an adapter is restarted after a configuration change, it reconnects to the platform automatically with the new credentials. This design allows configuration updates to take effect without any service interruption for the other platforms.

V. IMPLEMENTATION

PromptBridge is implemented in Node.js 18 or later. Each platform adapter is self-contained in its own directory and only imports the BotContext interface from the shared core module. Table I lists the SDK and connection method used by each adapter.

TABLE I

TABLE I. Supported Messaging Platforms

Platform	SDK	Ver.	Connection
Telegram	Telegraf	v4	Long polling
Discord	discord.js	v14	WebSocket
Slack	@slack/bolt	v4	Socket Mode
Teams	botbuilder	v4	HTTP adapter
GitHub	@octokit/rest	v22	Polling (120s)
Email	imapflow / nodemailer	v1/8	IMAP IDLE

The dispatcher imports no platform SDK. This is enforced by keeping all platform SDK imports inside the individual adapter modules. Adding a new platform therefore does not risk introducing a dependency conflict or an accidental coupling in the shared dispatcher code. The dispatcher has a single entry point for each type of incoming event: a text message, a file attachment, a callback action from a button, or a slash command.

Configuration is loaded at startup from a combination of environment variables and a JSON settings file. Environment variables handle bootstrap-only values such as the dashboard port and the dashboard bearer token. All runtime configuration, including platform tokens, agent executable paths, and allowlists, is stored in the JSON settings file and managed through the web dashboard. The configuration module caches settings for a few seconds and reloads them automatically, so dashboard changes propagate quickly without a restart.

Concurrency is handled at the level of individual conversations. The dispatcher keeps track of which conversations have an active run in progress. If a user sends a second command before the first run finishes, the dispatcher sends a brief message explaining that a run is already in progress and the new command will not start. This prevents multiple simultaneous runs on the same project from causing conflicting file changes, which would make the git diff report confusing and potentially corrupt the working tree.

Sessions are stored per chat ID and project ID. When the same user sends a message in the same conversation for the same project, the dispatcher loads the previous session and passes the stored history to the agent. This allows the agent to remember earlier instructions across multiple turns and build on previous results without the developer having to repeat context in every message.

File attachments received on any platform are downloaded to a local directory before the agent run begins. The dispatcher builds an enriched prompt that names each file and gives its saved path on disk. The agent can then read, modify, or reference those files using its normal file access, the same way it works with any project file. After the run, any new or changed files are uploaded back to the platform as attachments in the same conversation where the command was received.

A packaged Electron desktop application is provided for Windows, macOS, and Linux. The Electron app wraps the same Node.js server code and communicates with it through a secure IPC bridge. Data is stored in the OS user data folder, which means no manual path configuration is needed on any platform. The app includes a small control window showing the status of each platform adapter with

buttons to start or stop individual adapters without stopping the whole application. An icon in the system notification area lets the app run invisibly in the background.

The REST API exposed by the web dashboard has endpoints for all major operations. Clients can list and manage projects, view and delete sessions, read and update settings, and control platform lifecycle. The /api/status endpoint is public and returns the running status of each platform adapter, the software version, and the number of active runs. All other endpoints that change state require the bearer token. This makes it straightforward to integrate PromptBridge with external monitoring or automation tools.

Platform adapters can be started and stopped independently through the API without affecting other running adapters or active runs. When the dashboard updates a token for a platform that is currently running, the platform adapter is automatically restarted with the new configuration. The /api/run/email endpoint allows an external system such as a CI pipeline or a scheduled task to trigger an agent run directly and receive the result by email, without going through any messaging platform.

VI. EVALUATION

A. Extensibility

Adding a new platform requires writing a BotContext adapter and registering it in the startup sequence. The adapter must implement all BotContext methods: sendMarkdown, sendText, editMessage, showTyping, sendFile, sendWithButtons, acknowledgeAction, and updateButtonMessage. Once registered, the new platform receives commands and sends results using the exact same dispatcher code that all six existing platforms use. No other file needs to change.

Each existing adapter has between 160 and 220 lines of platform-specific code. The shared dispatcher has about 430 lines. This means roughly two-thirds of the functional logic is shared across all platforms rather than duplicated. When a bug is found in the dispatcher, fixing it once fixes the behavior on all six platforms simultaneously.

Adding a new agent requires only a new runner function. A minimal runner must build the correct command-line invocation for the agent, forward output lines back to the dispatcher as they arrive, and return a list of files changed when the process exits. A complete runner that handles all of these steps can be written in under 80 lines. No changes are needed in any platform adapter or in the dispatcher.

B. Comparison with Related Work

Table II compares PromptBridge to five related tools across six functional dimensions. Agent frameworks like LangChain and AutoGen are built for composing logic between model calls and tool invocations. They do not have a built-in way to receive tasks from a messaging platform or to push results back into a chat channel. They could be combined with a platform SDK, but that combination would still need a custom integration layer for each platform, which is exactly what PromptBridge provides as a ready-to-use system. Bot frameworks like Rasa and Dialogflow handle messaging and dialogue management well but have no integration with coding agents or file systems. They are designed for natural language understanding in customer service settings, not for running developer tools. Platform-specific SDKs like Slack Bolt are tied to a single platform and require all workflow logic to be duplicated if the team uses more than one messaging tool.

TABLE II *Comparison with Related Tools*

Feature	PromptBridge	LangChain	AutGen	Slack Bolt	Rasa
Coding agent integration	Yes	Partial	Partial	No	No
Multi-platform support	Yes (6)	No	No	No (1)	No
Shared workflow logic	Yes	Yes	Yes	No	No
Messaging delivery	Yes	No	No	Yes	Yes
Git change reporting	Yes	No	No	No	No

No public server required	Yes	No	No	No	No

C. Example Workflows

Three examples illustrate the system in practice.

Remote code review: A developer using only a phone sends /claude checks the failing tests in Slack during a commute. PromptBridge looks up the active project for that Slack channel and user, starts Claude Code in the project directory with the developer's stored session history, and sends a progress indicator back to the thread. When the run finishes, PromptBridge posts the findings and a list of changed files back to the same Slack thread. The developer can read the results and reply with a follow-up question without ever opening a terminal.

GitHub comment automation: A team lead reviewing a pull request adds /codex clean up the error handling in this module as a comment on the PR. Within two minutes, the polling adapter detects the comment, runs Codex on the linked project, and posts the agent's output as a reply to the same pull request comment. No webhook server is needed. The team lead sees the result in the same GitHub interface they were already using.

Email-based task: A developer sends an email with the subject Project: api-server and the body hi /claude summarize the changes merged this week. The IMAP listener picks up the email within a few seconds, matches the project from the subject line, runs Claude Code on that project with the requested prompt, and replies to the email thread with the summary. Any files the agent created are attached to the reply email.

D. Response Time Characteristics

The time from sending a command to receiving a result depends mainly on how long the AI coding agent takes to run, not on PromptBridge itself. The framework adds roughly one to two seconds of overhead per run for session loading, process startup, and result formatting. For real-time platforms like Slack, Telegram, and Discord, this overhead is not noticeable against the background of typical agent run times, which range from a few seconds for simple questions to several minutes for large refactoring tasks. Progress updates are streamed back to the user during the run, so the channel shows activity from the start and the developer is not left waiting without feedback.

The GitHub polling adapter has a different latency profile. Because it checks for new commands only every 120 seconds, the maximum delay from posting a command to the agent starting is just under two minutes. This is acceptable for the asynchronous nature of pull request and issue discussions but would not be suitable for interactive tasks where the developer is waiting actively. For interactive use cases, Slack or Telegram should be preferred because they deliver messages in near real time.

E. Limitations

The system depends entirely on the underlying agent behaving correctly. It does not sandbox agent execution, so an agent can read and modify any file in the configured project directory. This is acceptable in personal and small-team settings but would need stronger isolation controls in a shared environment with many users. The GitHub polling interval of 120 seconds is a hard lower bound set by the GitHub Search API rate limit, which makes the GitHub adapter less suitable for time-sensitive tasks compared to real-time platforms like Slack or Telegram. The Email adapter does not support button menus because email protocols have no mechanism for structured interactive inputs. Conversation history also grows without bound unless the user manually clears it with the /session command, which may eventually lead to very long prompts being passed to the agent.

VII. DISCUSSION

The main practical benefit of PromptBridge is that it meets developers where they already are. Rather than asking a developer to open a terminal and remember the correct command syntax, the system accepts a plain message in the tool the developer is already using. This reduces the barrier to using AI coding agents from a multi-step context switch to a single message in an existing conversation. The lower barrier means agents are more likely to be used for smaller tasks, not just large planned efforts.

The git-based change summary is one of the features that users find most useful. Knowing exactly which files the agent modified after a run gives the developer

a clear starting point for review. When working remotely or on a mobile device, this is much more useful than reading through the full agent output to figure out what changed. The developer can look at the file list, pull the latest changes in their IDE when they return to a workstation, and review only the files that were touched.

Polling was chosen over webhooks for GitHub specifically because many development environments cannot expose a public URL. A webhook listener requires a publicly reachable server address, which involves DNS configuration, firewall rules, and often a cloud server or reverse proxy. Polling requires only a personal access token with repository read and write permission. This makes the GitHub integration genuinely accessible to individual developers and small teams without any infrastructure setup.

The BotContext interface is the most consequential design decision in the system. By making the platform adapter responsible for translating all platform-specific details into a fixed set of method calls, the dispatcher becomes completely independent of any specific platform. This means the dispatcher can be tested in isolation using a mock adapter, which is much simpler than testing against a live Slack or Telegram connection. It also means any bug or improvement to the dispatcher benefits all six platforms at once with no additional effort.

Security is the most important area for future work. The current allowlist approach trusts every user on the list to run agents with full access to the project directory. A production-grade version of the system should run each agent inside a container or sandbox that limits which files and system calls the agent can access. Per-command confirmation prompts for destructive operations and audit logging of all agent runs would also strengthen the security posture significantly for team deployments.

Output formatting is another area with room for improvement. Agents produce markdown output that renders well in Telegram, Slack, and Discord, all of which support markdown natively. Microsoft Teams and Email both have limited markdown support, so code blocks, headings, and bullet lists may render as plain text. Post-processing the agent output to convert markdown to platform-native formatting, such as HTML for email or Adaptive Cards for Teams, would improve readability significantly on those platforms.

The relationship between session history and agent context windows is worth noting. As a conversation grows through many turns, the stored session history grows with it. When the history approaches the context window limit of the underlying model, the agent may begin to drop earlier parts of the conversation. A future version of PromptBridge could automatically summarize older session turns before passing the history to the agent, keeping the most recent context while staying within model limits.

From a software engineering perspective, PromptBridge demonstrates that a thin integration layer built around a well-defined interface can bridge two very different ecosystems: the world of messaging platforms, which are designed for human conversation with real-time delivery and interactive elements, and the world of coding agents, which are designed for file system access and long-running process execution. The BotContext interface is the boundary between these two worlds, and keeping that boundary clean is the central architectural insight of the system.

Looking ahead, the most impactful extension to the system would be a user study with real software development teams. The current design choices, such as polling for GitHub, buffering email output, and caching settings with a short TTL, were motivated by practical reasoning about developer workflows. Controlled experiments measuring actual task completion time, error rates, and developer satisfaction would validate or challenge these choices and provide a stronger empirical basis for future design decisions about multi-platform agent integration systems.

VIII. CONCLUSION

PromptBridge connects six messaging platforms to four AI coding agents through a four-layer architecture with a single shared dispatcher. When a developer sends a task in any supported channel, the system identifies the active project, runs the appropriate agent, and sends the result back to the same conversation. Conversation history and file attachments are handled automatically across all platforms. After each run, the developer receives a list of files that changed, giving them a clear starting point for review.

The four-layer design keeps platform-specific code isolated from workflow logic. Each platform adapter has between 160 and 220 lines of code that are entirely

self-contained. The dispatcher handles all six platforms in approximately 430 lines with no platform-specific branches. A new platform can be added by implementing the BotContext interface in a new adapter directory and registering it at startup. A new agent requires only a new runner function with no changes to any adapter or to the dispatcher.

The system runs on Node.js 18 or later and is also packaged as a desktop application for Windows, macOS, and Linux. No public server or cloud account is required. The GitHub integration works using only a personal access token with polling, making it easy to set up even on a developer laptop on an internal network.

From an ecosystem standpoint, PromptBridge is positioned between the agent frameworks that provide execution intelligence and the platform SDKs that provide message delivery. It does not compete with either. It depends on the agents for the hard part, which is understanding code and producing useful changes, and it depends on the platform SDKs for the delivery layer. Its value is in connecting them through a protocol that preserves context across turns and reports what actually changed after each run.

Future work should focus on three main areas. First, evaluating the system with real software teams over extended periods would provide concrete measurements of how much context switching is reduced and how this affects developer productivity and satisfaction. Second, adding container-based execution isolation would make the system safer for deployment in larger shared environments. Third, improving output formatting for platforms that do not support markdown natively, and expanding support to additional messaging platforms such as WhatsApp and Matrix, would increase the reach of the system given the straightforward extension model the current architecture provides.

REFERENCES

[1] M. Chen et al., "Evaluating Large Language Models Trained on Code," arXiv preprint arXiv:2107.03374v2, Jul. 2021.

[2] A. Ziegler et al., "Productivity Assessment of Neural Code Completion," in Proc. 6th ACM SIGPLAN Int. Symp. Machine Programming (MAPS), San Diego, CA, USA, Jun. 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, Dec. 2022, doi: 10.1126/science.abq1158.

[4] Anthropic, "Claude Code," Anthropic, 2025. [Online]. Available: https://www.anthropic.com/claude-code. [Accessed: Jul. 14, 2026].

[5] Anysphere Inc., "Cursor AI Code Editor," Cursor, 2025. [Online]. Available: https://cursor.com/. [Accessed: Jul. 14, 2026].

[6] C. E. Jimenez et al., "SWE-bench: Can Language Models Resolve Real-World GitHub Issues?," arXiv preprint arXiv:2310.06770, 2023.

[7] J. Yang et al., "SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering," arXiv preprint arXiv:2405.15793, 2024.

[8] Cognition AI, "Introducing Devin, the First AI Software Engineer," Mar. 2024. [Online]. Available: https://www.cognition.ai/blog/introducing-devin. [Accessed: Jul. 14, 2026].

[9] S. Yao et al., "ReAct: Synergizing Reasoning and Acting in Language Models," arXiv preprint arXiv:2210.03629, 2022.

[10] Q. Wu et al., "AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation," arXiv preprint arXiv:2308.08155, 2023.

[11] LangChain, "LangChain Documentation." [Online]. Available: https://python.langchain.com/. [Accessed: Jul. 14, 2026]..

[12] LlamaIndex, "LlamaIndex Documentation." [Online]. Available: https://docs.llamaindex.ai/. [Accessed: Jul. 14, 2026].

[13] Google Cloud, "Dialogflow Documentation." [Online]. Available: https://cloud.google.com/dialogflow/docs. [Accessed: Jul. 14, 2026].

[14] A. Bocklisch et al., "Rasa: Open Source Language Understanding and Dialogue Management," arXiv preprint arXiv:1712.05181, 2017.

[15] Slack Technologies, "Bolt for JavaScript Documentation." [Online]. Available: https://tools.slack.dev/bolt-js/. [Accessed: Jul. 14, 2026].

[16] Microsoft, "Bot Framework SDK Documentation." [Online]. Available:

https://learn.microsoft.com/azure/bot-service/. [Accessed: Jul. 14, 2026].

[17] T. D. LaToza et al., "Maintaining Mental Models: A Study of Developer Work Habits," in Proc. 28th Int. Conf. Softw. Eng. (ICSE), 2006, pp. 492-501, doi: 10.1145/1134285.1134355.

[18] C. Bird et al., "Does Distributed Development Affect Software Quality?," in Proc. 31st Int. Conf. Softw. Eng. (ICSE), 2009, pp. 518-528, doi: 10.1109/ICSE.2009.5070547.

[19] D. Hilton et al., "Usage, Costs, and Benefits of Continuous Integration," in Proc. ASE, 2016, pp. 426-437, doi: 10.1145/2970276.2970358.

[20] J. Liu et al., "Large Language Model-Based Agents for Software Engineering: A Survey," arXiv preprint arXiv:2409.02977, 2024.

[21] E. Nijkamp et al., "CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis," arXiv preprint arXiv:2203.13474, 2022.

[22] Z. Guo et al., "DeepSeek-Coder: When the Large Language Model Meets Programming - The Rise of Code Intelligence," arXiv preprint arXiv:2401.14196, 2024.