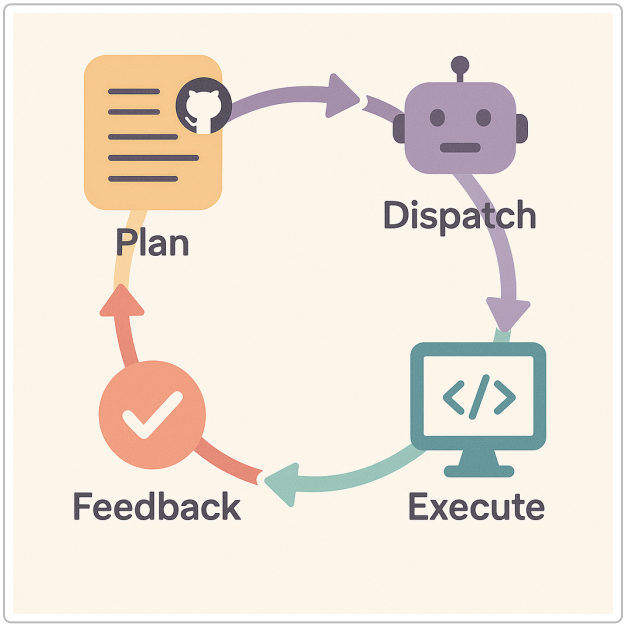
ChatGPT

Blueprint: The Autonomous, Generative Development Pipeline



Core Components & Agent Roles

GitHub as the Source of Truth

- **GitHub Issues** – Issues are the *fundamental work unit*. They allow teams to plan, discuss and track bug reports, new features and ideas [1]. Issues can be created through the web UI or via APIs, and they can be broken down into **sub-issues** to represent hierarchical work; sub-issues let you create multiple levels of nested tasks to break large projects into manageable pieces [2]. Issues support metadata such as labels, milestones and assignees, and can be linked directly to pull requests [3].

- **GitHub Projects** – Projects provide an adaptable table, board or roadmap for planning and tracking work [4]. They integrate directly with issues and pull requests, allowing teams to group, filter and sort work. Projects support customizable fields and views, and changes to issue metadata are reflected immediately in the project board [5]. Built-in automations can update the status of items (e.g., moving an item to **Done** when an issue or pull request is closed or merged) [6].

- **GitHub Actions** – Actions enable event-driven automation. Workflows can be triggered by events such as `push`, `pull_request` or label events. For example, the following configuration causes a workflow to run when an issue is opened or when a label is added to an issue [7]. The `push` event triggers a workflow whenever someone pushes changes to the repository [8] —critical for restarting

the analysis phase after a pull request is merged. Actions can also interact with Projects via the GraphQL API to automatically add or update items.

Agents and Their Responsibilities

- **Jules CLI (Remote Executor)** – Jules is an asynchronous coding agent that runs tasks inside a secure, short-lived Ubuntu virtual machine [9]. The environment comes with a pre-installed toolchain for popular languages and frameworks. The command `jules remote new --repo <repo> --session "<prompt>"` creates a remote session that clones the repository and executes the given task [10]. Jules is ideal for *mechanical or predictable tasks* such as dependency bumps, test execution or documentation updates; it can run in parallel to human work and reports results back via code changes or artifacts.

- **Aider CLI (Local Pair-Programmer)** – Aider is a local AI pair-programming agent that runs within the developer's repository. It maps the codebase, supports dozens of languages and integrates tightly with git. Whenever Aider edits a file, it commits the change with a descriptive commit message [11], making it easy to review or undo AI edits. Aider automatically commits its changes and can be configured to create a git repository if one doesn't exist [11]. Because of this fine-grained git integration, Aider excels at exploratory or complex tasks where a human might iteratively refine the solution.

- **Copilot CLI (Local Orchestrator)** – GitHub Copilot CLI is an interactive terminal agent that can run local commands or **delegate** work to the remote Copilot coding agent. Using the `/delegate` command followed by a prompt sends the current session (including file context) to the remote Copilot coding agent [12]. Copilot commits any unstaged changes as a checkpoint to a new branch, opens a draft pull request, makes changes in the background and requests a review [12]. Copilot CLI can therefore orchestrate tasks locally and offload medium-complexity code changes to the cloud.

- **Web Browser Agent (Researcher)** – A lightweight agent for external information gathering. It is used when issues are labeled `research` and is responsible for finding documentation, investigating error messages or comparing alternative libraries. Results are posted back to the corresponding GitHub issue as a comment. Because it operates outside the codebase, the web agent can access up-to-date documentation and web resources to inform the development process.

The Data Model (Source of Truth)

Artifact	Purpose	Key Details
Issue (Epic/ Story/ Task)	Fundamental unit of work. Issues track bugs, enhancements, tests or documentation tasks [1].	Each issue may include a description, labels (e.g., `bug`, `tech-debt`, `jules-task`), assignees and milestones. Hierarchical **sub-issues** allow a parent issue to be broken down into up to eight levels of nested tasks [2].

Artifact	Purpose	Key Details
Project Board	Flexible planning board integrated with issues and pull requests [4] .	Columns represent states (e.g., **Triage**, **Ready-for-Agent**, **In Progress**, **Review**, **Done**). Projects maintain direct references to the issues and pull requests they display; status changes propagate bidirectionally [5] . Built-in automations update statuses when items are added, closed or merged [6] .
Plan file (YAML/ JSON)	Machine-readable representation of analyzer output.	The analyzer agent produces a plan file capturing the list of tasks it has identified (e.g., location in code, recommendation, severity, suggested labels). This file allows the dispatch process to create new issues programmatically and to avoid duplicates by checking existing issues.

The Autonomous Workflow (Step-by-Step)

Phase 1 – Generative Planning & Triage (Codebase Analysis)

1. **Trigger:** A scheduled GitHub Action or a `push` to the default branch after a pull request merge. Push events fire whenever someone pushes changes to the repository [8] .
2. **Analyzer Agent Execution:** A script (e.g., using Copilot CLI or custom static-analysis tools) scans the source and test directories. It looks for `TODO` comments, linter warnings, uncovered functions, obsolete dependencies, security advisories or undocumented public APIs. The analyzer may leverage language-specific tools (e.g., ESLint, PyLint, Snyk) and pattern matching.
3. **Plan Creation:** For each finding, the analyzer writes an entry into a machine-readable plan file. Each entry includes a short title, description, file/line context, severity and recommended labels (e.g., `bug`, `tech-debt`, `docs`, `jules-task`).
4. **Issue Generation:** The analyzer uses GitHub's API to create a new issue for each task. Sub-issues are created for hierarchical breakdowns (e.g., an epic issue for "Add integration tests" with sub-issues for each module). Issues are assigned appropriate labels and added to the **Triage** or **Backlog** column in the project board.

Phase 2 – Agent Dispatch & Tasking (Work Assignment)

1. **Trigger:** A human triage (product owner or lead developer) or automation moves an issue from **Triage** to **Ready-for-Agent** on the project board.
2. **Event Detection:** A GitHub Action listening for project card movements or issue label events (`issue_labeled`) is triggered [7] . The action reads the issue's labels and metadata to determine which agent should handle it.
3. **Decision Logic:**
4. **Scenario 1 – Jules:** If the issue is labeled `jules-task` (e.g., "Update all npm dependencies"), the action runs `jules remote new --repo <repo> --session "<issue title>"` [10] . The session prompt can include a link to the issue description. Because Jules runs inside a secure VM and has a preinstalled toolchain [9] , it is suited for mechanical tasks such as dependency bumps, test runs or automated documentation generation.

5. **Scenario 2 – Copilot:** If the issue is labeled `copilot-delegate`, the action invokes Copilot CLI and uses the `/delegate` slash command with a prompt summarizing the issue. Copilot commits any unstaged changes, creates a new branch and opens a draft pull request [12]. The remote Copilot coding agent then makes the modifications autonomously.
6. **Scenario 3 – Aider:** If the issue is labeled `aider-interactive` or requires complex reasoning, the issue is assigned to a human developer. The developer runs Aider CLI locally, adding the relevant files to the chat session and iterating until the task is complete. Aider commits each change with descriptive messages [11], allowing peer review.
7. **Scenario 4 – Research/Web:** If the issue is labeled `research`, the Web Browser Agent is triggered. It uses internet search to collect external documentation or compare libraries and posts the findings as a comment on the issue. The issue then returns to **Triage** with updated information.

Phase 3 – Execution & Pull Request (Modification)

1. **Work Execution:** The selected agent performs the task.
2. *Jules* executes tasks inside its VM, using the repository context and any specified setup scripts. It produces code changes, documentation updates or test results.
3. *Copilot coding agent* works on the new branch in the remote environment, generating code, tests or documentation. It maintains context and can continue iterating via interactive prompts.
4. *Aider* operates locally with the developer in control, producing commits for each incremental change.
5. *Research* tasks produce comments only; no code changes.
6. **Pull Request Creation:** For code-changing agents, a pull request is opened automatically. Jules or Copilot attach the new branch to a draft or open PR, and Aider's developer opens a PR manually. The PR references the original issue (e.g., using `closes #123`) so that merging will close the issue [3]. The project card moves to the **In Review** column via built-in automation.
7. **Review & Merge:** A human reviewer or automated tests evaluate the pull request. After approval and passing checks, the PR is merged into the default branch. Built-in project automations mark the item as **Done** when the issue or PR closes [6].

Phase 4 – The Generative Feedback Loop (Real CI/CD)

1. **Trigger on Merge:** Merging the pull request creates a new commit on the default branch, causing the `push` event to fire [8]. This triggers the Phase 1 analysis workflow again.
2. **Continuous Improvement:** The analyzer scans the updated codebase, capturing new `TODO`s or consequences of the recent change (e.g., new functions without tests or documentation). It generates new issues and adds them to **Triage**, thus closing the loop.
3. **Cyclical Operation:** Each cycle enriches the backlog with fine-grained tasks and ensures the codebase continuously improves. Because agents operate autonomously and the pipeline is triggered by repository events, the system functions like a self-sustaining generative CI/CD pipeline.

Key Agent Interactions & Handoffs

1. **Analyzer → Dispatch:** The analyzer produces a plan file and issues. It attaches labels that the dispatch process uses to select an agent. Duplicate issues are avoided by checking existing issues before creation.
2. **Dispatch → Agent:** The GitHub Action reads the issue labels and chooses between Jules (`jules-task`), Copilot (`copilot-delegate`), Aider (`aider-interactive`), or Web Agent (`research`). Appropriate commands are executed to start the session. The issue moves to **In Progress**.

3. **Agent → GitHub:** Jules and Copilot produce code on separate branches and open pull requests referencing the issue. Aider commits changes locally; the developer opens a PR. Web Agent posts a comment. The issue status moves to **Review**.
4. **Merge → Analyzer:** When the pull request merges, the project automation marks the issue **Done** [6], and the `push` event triggers a new analysis cycle [8]. This ensures continuous feedback and improvement.

Workflow Scenarios & Potential Bottlenecks

1. **Jules Task Fails or Times Out:** If a Jules session fails (e.g., dependency installation fails or tests do not pass), the session reports an error. The GitHub Action can detect this and automatically create a follow-up issue labeled `aider-interactive` for human intervention. Alternatively, it can retry the session with additional parameters.

2. **Analyzer Creates Duplicate Issues:** The analyzer could inadvertently create duplicate issues for the same TODO or code smell. Mitigation includes querying existing issues for matching file paths or titles before creating a new one. The plan file can record unique identifiers (e.g., a hash of file path + line number) to detect duplicates.

3. **Merge Conflicts Between Agents:** Multiple agents might work on overlapping code areas (e.g., Jules upgrades dependencies while Copilot refactors a module). When both branches are merged, conflicts may occur. GitHub Actions can detect conflicts and assign a new issue labeled `aider-interactive` for a human to resolve using Aider. Aider's git integration makes it easy to handle these conflicts interactively.

4. **Incorrect Agent Assignment:** An issue might be mislabeled, causing an inappropriate agent to handle it (e.g., Copilot trying to fix a security vulnerability better handled by a human). To mitigate this, the analyzer can suggest default labels based on heuristics, but triage by humans remains critical. Project board columns such as **Ready-for-Agent** give humans the opportunity to correct labels before dispatch.

5. **Long-Running or Resource-Intensive Tasks:** Some tasks (e.g., rewriting a large module) may exceed the time limits of Jules's VM or Copilot's session. These tasks can be broken into smaller sub-issues during the analysis phase or delegated to Aider for incremental implementation. Alternatively, the plan file can specify a `human-required` flag to route them directly to developers.

This blueprint outlines a high-level architecture for an autonomous, generative development pipeline. By combining GitHub's planning tools, automation features and AI-powered agents, the system continuously analyzes the codebase, plans work, executes tasks and feeds the results back into future planning. The result is a self-perpetuating CI/CD loop that not only integrates and deploys code but also drives continuous improvement.

---

[1] [2] [3] About issues - GitHub Docs
https://docs.github.com/en/issues/tracking-your-work-with-issues/learning-about-issues/about-issues

4 5 About Projects - GitHub Docs
https://docs.github.com/en/issues/planning-and-tracking-with-projects/learning-about-projects/about-projects

6 Using the built-in automations - GitHub Docs
https://docs.github.com/en/issues/planning-and-tracking-with-projects/automating-your-project/using-the-built-in-automations

7 Workflow syntax for GitHub Actions - GitHub Docs
https://docs.github.com/en/actions/reference/workflows-and-actions/workflow-syntax

8 Workflow Triggering Events and Event Actions - DEV Community
https://dev.to/kalkwst/workflow-triggering-events-and-event-actions-5cec

9 Environment setup | Jules
https://jules.google/docs/environment/

10 Jules Tools Reference | Jules
https://jules.google/docs/cli/reference

11 Git integration | aider
https://aider.chat/docs/git.html

12 Using GitHub Copilot CLI - GitHub Docs
https://docs.github.com/en/copilot/how-tos/use-copilot-agents/use-copilot-cli