ChatGPT

Implementation Plan for AutoDevHub (AI-Powered DevOps Tracker)

1. Project Setup and Environment

Objective: Establish the repository structure and development environment quickly, leveraging GitHub Codespaces and preparing AI integration.

- **Repository & Codespace:** Create a new repository (e.g., `ai-cohort-july-2025/AutoDevHub`) with the proposed directory layout. Initialize a GitHub Codespaces instance for a ready-to-code environment.
- *Directory Layout:* Set up folders for `backend/` (FastAPI app), `frontend/` (React app), `docs/` (documentation for GitHub Pages), `.github/workflows/` (CI/CD configurations), `presentation/` (slides and demo script), and a root `README.md` as described in the summary.
- **GitHub App for Claude:** Install the Anthropics **Claude GitHub App** on the repo and configure authentication. Add required secrets (e.g. `ANTHROPIC_API_KEY` or Claude OAuth token) to enable the **Claude Code Action** in workflows [1]. This provides Claude the ability to act on PRs and issues.
- **Dev Environment Tools:** Ensure the Codespace has Python (for FastAPI), Node.js (for React), and necessary CLI tools:
- Python dependencies: FastAPI, Uvicorn, SQLAlchemy/Pydantic (for models), PyTest, etc.
- Node/React: Use a fast setup like Vite or Create React App for the frontend.
- Mermaid support: If needed, install a Markdown preview or plugin to render Mermaid diagrams for verification.
- **Time Allocation:** *Estimated ~30-45 minutes.* This includes repo scaffolding, Codespace launch, and configuring secrets for AI action integration.

2. AI-Generated Documentation

Objective: Leverage AI (Claude) to quickly produce core project documentation from minimal input, ensuring comprehensive coverage of requirements and architecture.

- **Product Requirements Document (PRD):** Begin by prompting Claude with a one-line idea of the product (e.g., *"An AI-assisted Agile DevOps Tracker that auto-generates and deploys software artifacts"*). Instruct Claude to produce a structured PRD covering objectives, features, user stories, and success metrics [2]. The AI can analyze provided context to draft a comprehensive PRD, which saves time in the initial phase [2]. Review and tweak the PRD for accuracy and clarity (this can be iterative: ask Claude to refine sections as needed).
- **Architecture Document:** Ask Claude to generate an architecture overview in Markdown, including both high-level description and UML diagrams:
- Describe the system components (frontend, backend, CI pipeline) and their interactions. Have Claude produce a **component diagram** or **sequence diagram** in Mermaid syntax. Claude is capable of generating Mermaid.js diagrams from descriptions [3], so provide a prompt like: *"Generate a*

Mermaid diagram showing the workflow: user -> frontend -> backend -> database, and CI/CD automation." The resulting diagram code (e.g., a `graph LR` or sequence chart) is included in the architecture doc as a fenced code block with mermaid syntax.
- Include an architecture decisions section. Use Claude to create **Architecture Decision Records (ADRs)** for key choices (e.g., "Why FastAPI?", "Why React?", "Use of GitHub Actions for CI"). Each ADR should state the decision, alternatives, and rationale (Claude can draft these justifications).
- **ADR Documents:** For each significant decision (backend framework, frontend framework, CI approach, database choice), generate a short ADR file in `docs/adr/` (if using an ADR directory) or include them in the architecture doc. Claude can justify decisions (for example, choosing FastAPI for Python simplicity, or using GitHub Pages for documentation hosting).
- **Documentation Format:** Use clear Markdown structure with headings and bullet points. Ensure mermaid diagrams are properly fenced with ```mermaid for later rendering. Keep language concise and professional, with the AI providing polish.
- **Quality Check:** After generation, read through each document. Use Claude in "review" mode to improve clarity or fill any gaps. (For example, *"Review the PRD and suggest any missing functional requirements or risks."*)
- **Time Allocation:** *Estimated ~1.5 hours.* (PRD: 30 min, Architecture doc: 30-45 min, ADRs: 15-30 min). AI assistance significantly accelerates this phase, as drafting these documents manually could take much longer, but AI can generate a solid first draft within minutes [2].

3. Backend Implementation (Python FastAPI)

Objective: Rapidly develop a functional backend with database schema, REST API endpoints, and tests using AI-generated scaffolding.

- **Database Schema & Models:** Use Claude to design a simple SQL schema and corresponding ORM models. For instance, define tables for "UserStories" or "Features" if needed (though the core functionality is the story generator, a complex DB may not be necessary for the demo). Prompt Claude with: *"Generate a SQL schema for storing user feature requests and generated Gherkin stories."* Save the output to `backend/schema.sql`. Next, create `backend/models.py` with SQLAlchemy models or Pydantic models based on the schema (Claude can produce the Python classes and relationships).
- **FastAPI Scaffold:** Initialize a FastAPI app (e.g., `backend/main.py` or `backend/app.py`). Use Claude to generate boilerplate code for FastAPI:
- Include necessary `FastAPI()` app creation, CORS middleware (since a React frontend will call it), and a health-check endpoint.
- Key Endpoint – `/generate-story`**:** This POST endpoint will accept a feature idea (text) and return a generated user story in Gherkin format. Have Claude implement this function. It might use a prompt internally to call Claude's API or a stub logic (for demo, possibly just echo the input in a Gherkin template unless we actually integrate an AI call). Since we want to demonstrate AI, consider integrating a call to Claude or an open-source model for on-the-fly story generation if time permits; otherwise, simulate it.
- Example: The endpoint takes JSON `{"feature": "As a user, I want ..."}` and returns a multi-line string of a Gherkin feature with `Given/When/Then` clauses. Claude can generate an example response format for a known input.
- **Unit Tests (PyTest):** For each endpoint and model, use AI to create basic tests. Prompt Claude: *"Write pytest unit tests for the FastAPI app endpoints, including /generate-story (simulate a sample input and*

verify the response contains 'Given' or some expected text)." This yields tests in `backend/tests/test_endpoints.py` (and similarly test the model or any logic). Even if the tests are simple (e.g., using FastAPI's TestClient), they provide quick validation of the code. Generative AI has been shown to effectively draft unit tests, improving coverage and catching issues with minimal human effort [4].

- **Static Analysis & Security:** Run a quick security scan on the code:
- Use a tool like **Bandit** (for Python security linter) or **flake8** for style issues. Collect any findings.
- Ask Claude to create a `SECURITY_REPORT.md` summarizing potential vulnerabilities or improvements. For example, *"Analyze the FastAPI code for security issues (like missing input validation, or open CORS) and produce a brief security audit report."* Claude's analysis can spot things like lack of authentication on endpoints, etc., which you document. This fulfills the "static security scan report" requirement.
- **Iteration:** If tests fail or Bandit flags issues, quickly fix them. Claude can assist in fixes; e.g., *"Fix the CORS configuration and update tests accordingly."* Utilize Claude's capability to not only review but also implement code changes [5] to speed up iteration.
- **Time Allocation:** *Estimated ~2 hours.* (Schema/Models: 30 min, Endpoints: 30 min, Tests: 30 min, Security review & fixes: 30 min). Thanks to AI automation, generating boilerplate and tests is fast – AI agents can handle code editing and testing tasks autonomously [6], drastically cutting down development time.

4. Frontend Implementation (React)

Objective: Build a lightweight React front-end to interact with the backend, focusing on the Story Generator feature. Keep it simple for speed.

- **Bootstrap React App:** Use **Vite** or Create React App to quickly scaffold a new React project under `frontend/`. (Vite is preferred for speed). This setup (initial commit) can be done in a few minutes (using `npm create vite@latest`).
- **Component Creation – `StoryGenerator.jsx`:** Prompt Claude to implement a React component that provides:
- A text input or textarea for the user to enter a feature idea.
- A "Generate Story" button that triggers an API call to the FastAPI backend (`/generate-story` endpoint).
- Display area for the returned Gherkin story.
- Basic state management (using React `useState`) to store the input and output.
- Example prompt: *"Provide a React component named StoryGenerator that takes user input and on button click, calls fetch('/generate-story') to get a Gherkin story. Display the response."* Claude will output JSX code and possibly some CSS.
- **API Integration:** Configure the API URL. If the app is served on a different port, ensure CORS is handled (which we did in backend). For development in Codespaces, the backend might run on e.g. port 8000, front-end on 3000 – adjust fetch URL accordingly (maybe use a proxy or environment variable in development).
- **UI/UX Considerations:** Keep styling minimal due to time constraints. Perhaps use a simple form layout. Claude can generate CSS or inline styles if needed, but focus on functionality first. We can always prompt: *"Improve the UI of the StoryGenerator component with simple styling (centered form, nice button)."* if time permits.
- **Testing the Frontend:** Manually verify the component:

- Run the backend (Uvicorn) and `npm start` for React in the Codespace. Input a sample idea ("As a user, I want to save tasks...") and click the button – confirm that a Gherkin-format story appears.
- If any errors (CORS, JSON format, etc.), adjust quickly. (Use browser console to debug; fixes should be minor).
- **Time Allocation:** *Estimated ~1 hour.* (Initial scaffold: 10 min, Component coding via AI: 20 min, Integration & testing: 30 min). Using AI to write the component saves significant time on boilerplate, allowing focus on quick testing.

5. Continuous Integration & Automation with GitHub Actions (AI-Assisted)

Objective: Set up automated workflows to leverage AI in reviewing code, updating documentation, and deploying the docs site, aligning with Agile DevOps practices.

- **AI Code Review Workflow (`.github/workflows/ai-code-review.yml`):** Utilize the Claude Code GitHub Action to automate pull request reviews.
- **Trigger:** on pull_request (opened or updated).
- **Action:** Use `anthropics/claude-code-action@beta` in analysis mode. Configure it to read the diff of the PR and comment with suggestions. Claude's capabilities include analyzing PR changes and suggesting improvements in code quality or style [7]. It can function as an intelligent code reviewer, pointing out potential bugs or improvements automatically.
- **Test Verification:** Optionally, integrate this with running the test suite. For example, the workflow can have a job to run `pytest`. If tests fail, Claude could be prompted to analyze the failure (though this is advanced). At minimum, ensure that test results (pass/fail) are output so the developer can see them. If using Claude in a comment, we might skip asking it to interpret test output to keep within time.
- **Outcome:** Every PR should receive AI feedback. This ensures code quality without waiting for human review, and it aligns with the concept of an AI pair programmer reviewing code [5].
- **Automated Documentation Update Workflow (`update-docs.yml`):** Keep documentation in sync with code by using AI to auto-update the docs after changes.
- **Trigger:** on push to main (or merge to main).
- **Steps:** A possible approach: generate a summary of changes and have Claude update relevant sections in PRD or architecture docs.
    - Extract commit messages or diff summary (using `git log -1` or similar).
    - Prompt Claude: *"Update the documentation (PRD.md, architecture.md) to include the following new feature or changes: [list of changes]. Ensure consistency with the current docs."* Claude can read the current docs from the repository (the action has read/write access to contents [8]) and apply edits.
    - Save the modified docs and commit them back to the repo (the action can push changes or open a PR with updates).
- **Diagrams:** If architectural changes are significant (e.g., a new component added), instruct Claude to adjust or regenerate the Mermaid diagrams. This might be done in a separate prompt or as part of the above step.
- **Outcome:** The documentation (PRD, architecture, ADRs) evolves alongside the code, with minimal manual effort. This dynamic updating ensures the docs site is always up-to-date with latest features – a big win for project consistency.

- **GitHub Pages Deployment (`pages-deploy.yml`):** Automate publishing of the `docs/` folder to GitHub Pages for a live documentation site.
- **Setup:** In repository settings, enable GitHub Pages to serve from the `docs/` directory on the main branch (or use a workflow to deploy to `gh-pages` branch). Given the simplicity (Markdown docs with Mermaid), we can use GitHub's native Jekyll or a static site generator if time permits. A straightforward approach is:
    - Trigger on push (or on documentation update).
    - Use an action like `actions/upload-pages-artifact@v1` and `actions/deploy-pages@v1` (GitHub's official way to deploy pages from workflows).
    - The docs likely don't need a build step (unless we convert Mermaid to images for viewing – but we can use a client-side approach to render mermaid by including the mermaid script in pages if needed).
- **Outcome:** The site (e.g., `https://username.github.io/AutoDevHub/`) will display the PRD, architecture doc (with diagrams), ADRs, and any other info in the `docs/` folder. This provides a polished touch to the capstone, showing a live project portal.
- **Secrets & Permissions:** Ensure the workflows have permission to read/write contents and issues. The Claude action needs the `ANTHROPIC_API_KEY` secret (or Claude OAuth token) which was configured earlier. Also give Pages deploy the necessary token (Pages build uses a special token by default).
- **Time Allocation:** *Estimated ~1 hour.* (Writing YAML workflows with AI help: 30-40 min, configuring any leftover settings: 20 min). We will draft one workflow at a time, possibly using examples from Claude's repo or documentation as a starting point. Testing each workflow (trigger a dummy PR to see if Claude responds, push a commit to see if docs update) can be done quickly within this time.

6. Final Presentation and Demo Preparation

Objective: Synthesize the project outcome into a clear presentation and a plan for a live demo, using Markdown for easy maintenance and potential conversion to slides.

- **Slide Deck (Markdown):** Create `presentation/slides.md` as a Markdown-based slide deck. Given the requirement for a Markdown-focused presentation, consider using a tool like *Marp* or *reveal.js* markdown to later present, but initially just focus on content:
- Include an introduction slide (project title, objective), slides for each major section (Documentation, Backend, Frontend, CI/CD Automation), and a conclusion slide (lessons learned or future work).
- Use bullet points and short statements, not paragraphs, to fit a slide format. Highlight how AI was used in each stage (e.g., "AI-generated PRD", "Claude code review on PRs", etc.).
- If possible, embed the UML diagrams or code snippets as images or fenced code for clarity. (We can later use a conversion tool to turn this MD into a slide deck PDF or HTML, if desired).
- **Demo Script:** In `presentation/demo_script.md` (or integrated with slides notes), outline the live demo steps:
- **Intro (1-2 min):** Explain the project goal and the innovative AI-driven approach.
- **Docs Showcase (2 min):** Open the GitHub Pages site or the `docs/` markdown to show the PRD and architecture diagram. Emphasize these were AI-generated swiftly.
- **Feature Demo (3-4 min):** Run through the Story Generator feature. In the running app (or via an API call in Swagger UI/Postman), input a sample idea and show the Gherkin output. This proves the backend and frontend integration.

- **AI Workflows (2-3 min):** Simulate a code change to show off automation – for example, create a small PR (maybe tweak the README or add a minor feature) and show Claude's PR review comment appearing. Or show the contents of a PR where Claude left suggestions. Also, show how a commit triggers the docs update (perhaps by pointing to a recent commit that did so, or showing the Actions logs).
- **Conclusion (1-2 min):** Summarize the speed and efficiency gains: all major artifacts were produced in hours, thanks to AI. Mention any future extensibility (like possibly adding a planning agent or more features if given more time).
- **Rehearsal:** Since the target is 10–15 minutes, ensure the script timing aligns (~10 minutes content + buffer for QA if needed). Practice the demo to avoid hiccups (especially test the live feature beforehand).
- **Time Allocation:** *Estimated ~1 hour.* (Slides content: 30 min, Demo script: 20 min, Dry run: 10 min). Using Markdown for slides makes editing quick. We focus on the content, as visuals (diagrams, etc.) are largely already prepared in the docs. If any diagram or graphic is missing, we can quickly generate it (Claude can create illustrative ASCII art or simple diagrams if needed, but likely not necessary).

7. Timeline & Execution Strategy

To complete this capstone in **8 hours or less**, we will tightly manage time for each phase, using AI to accelerate tasks that normally take much longer:

1. **Initial Setup (0:00 - 0:45):** Repository, Codespaces, and Claude integration ready. *(~45 min)*
2. **Documentation Drafting (0:45 - 2:15):** AI-generated PRD, Architecture doc with diagrams, ADRs. *(~1.5 hours)*
3. **Backend Development (2:15 - 4:15):** FastAPI endpoints, database, tests, security review – heavily AI-assisted coding. *(~2 hours)*
4. **Frontend Development (4:15 - 5:15):** React app scaffold and integration with backend. *(~1 hour)*
5. **CI/CD Automation (5:15 - 6:15):** Implement GitHub Actions for AI code review, docs updates, and Pages deploy. *(~1 hour)*
6. **Presentation Prep (6:15 - 7:15):** Create slides and demo script in Markdown. *(~1 hour)*
7. **Buffer & Testing (7:15 - 8:00):** Use remaining time (~45 min) for any overflow in earlier tasks, final testing, and refinements. Ensure everything (docs, app, workflows) is working as expected.

Throughout each phase, **Claude (AI)** is embedded in the workflow to speed up output: - It generates drafts that humans typically refine, thus compressing work that could be several days into minutes (for example, drafting a full PRD from a one-liner idea) [2] . - It creates code and even tests, acting like an autonomous developer agent writing and verifying code [6] . - It reviews code and maintains quality, acting as an ever-present reviewer on PRs [7] . - It updates documentation continuously, so we don't spend time manually syncing docs with code changes.

By leveraging these capabilities, this plan ensures that a robust prototype of **AutoDevHub** can be implemented in a single working day. Each step is designed to showcase the power of AI in the software engineering lifecycle – from requirements to deployment – aligning with the project's goal of an AI-driven development workflow.

8. Conclusion

In summary, the implementation plan follows an agile, AI-first approach to building **AutoDevHub**. By breaking the project into clear phases and using Claude to automate and assist in each phase, we ensure full coverage of the capstone requirements within 8 hours. This not only results in a functioning AI-powered Agile DevOps Tracker, but also demonstrates the efficiency gains of AI-driven development, echoing research that AI agents can autonomously handle coding, testing, and documentation tasks in a secure environment [6]. The delivered project will include comprehensive AI-generated documentation, a working backend/frontend, continuous integration workflows with AI reviews, a published documentation site, and a prepared presentation – all evidence that modern development can be accelerated dramatically with the right AI tools and planning.

---

[1] [5] [7] [8] GitHub - anthropics/claude-code-action
https://github.com/anthropics/claude-code-action

[2] Using AI to write a product requirements document (PRD) | ChatPRD Resources
https://www.chatprd.ai/resources/using-ai-to-write-prd

[3] Generating AWS Architecture Diagrams with Claude - Micah Walter
https://www.micahwalter.com/2024/11/generating-aws-architecture-diagrams-with-claude/

[4] How to Use AI to Automate Unit Testing with TestGen-LLM and Cover-Agent
https://www.freecodecamp.org/news/automated-unit-testing-with-testgen-llm-and-cover-agent/

[6] [2403.08299] AutoDev: Automated AI-Driven Development
https://arxiv.org/abs/2403.08299