Helix CLI Specs

The Helix CLI is the AI coding agent built for the terminal to access and work (code, test, debug, etc.) with LLMs. Primary provider is LLama CPP, however it supports all other options such as work with remote APIs - DeepSeek, Qwen, Claude, Gemini, Grok, Mistral, etc. It supports various other providers like OpenRouter, Ollama, Nvidia, HuggingFace, etc.

It is powerfull programming tool which intracts with end user and makes possible flawless code implementation on the projects, refactopring, debugging, etc.

Dynamic hardware analysis and optimization

The CLI makes possible for end users to build and run local LLMs and use them for code generation, testing, debugging, etc. However, most important is that Toolking, Thinking and other LLM capabilities may be flwalessly used on low end computer systems (by using 7B LLMs). On more capable machines, dynamically detrmined, more powerful LLMs may be used - 32b and similar.

Architecture

All project / application components are fully decoupled programs which are bound together with proper bash scripts. Each comonent can be used separately or as a part of larger application bound together by bash scripts. All programs are writtent in Go language.

Some of the components that we are mentioning now are:

- LLama CPP API for communication with LLMs executed locally by LLama CPP
- OpenRouter API for communication with OpenRouter
- Ollama API for communication with Ollama
- DeepSeek API for communication with DeepSeek
- Qwen API for communication with Qwen
- Claude API for communication with Claude
- Other APIs for communication with other LLMs
- ASII art generator - will generate the header for the project that will appear in CLI UI when user starts the application
- Pure CLI mode (Headless CLI mode)
- CLI UI mode
- REST API mode (run as server)
- Postgres database component (run as server) - will be used by CLI UI and REST API, storing memories, progres, etc. Postgres database will be SQL Cipher protected!

- Debugger component - make sure that debugging of software for issues detection is possible using LLMs. It investigates and collects information about software execution for further fixes and improvements or new features development.
- Planner - Does the deep analysis and perfroms the planning with written documentation as result of this efforts - user inputs are required as always
- Builder - Builds the code using LLMs, extends the features, fixes issues, etc.
- Refactorer - Refactors the code using LLMs, extends the features, fixes issues, etc.
- Tester - Tests the code using LLMs, extends the features, fixes issues, etc. Tester introduces to the codebase of single or multiple modules of the projects the full 100% code coverage with several testing types: unit, integration, full automation, end-to-end, etc. Last two types will always be executed on real devices or emulators or apps ran by real AI QA! There are two more kind of tests that will always be performed: SoanrQube deep scanning and Snyk. Both performed with dockerized (docker compose) free versions of the two tools. Full reports will be generated and any discovered issues fully fixed.

LLama CPP and Ollama support

Under the hood API will contain its local instance (codebase) of LLama CPP and Ollama. It will be used for communication with LLMs via exposed API. Once it is ready it will be used for communication with LLMs. To be ready it has to download source code, do optimal build for current OS, do the configuration and optimization. API will expose downloadable LLMs for both LLama CPP and Ollama. It will download LLMs from all possible sources such as HuggingFace, OpenRouter, DeepSeek, Qwen, Claude, etc. Downloaded LLMs will be converted to proper format if that is needed so LLama CPP and Ollama will be able to use them. User will trigger the installation of the LLM and it will be ready for use. Installation will first make sure to detect hardware of host machine - all mandatory capabilites, then it will build local project instance of LLama CPP and Ollama and will download the LLM to be used. All optimization performed and LLM downloaded and started.

All utils have to be decoupled and other components for installation, configuration, download and running the LLMs. Make sure we have proper generics and interfaces which are widely reusable! Resusability principle has to be followed with every project component!

Others APIs - VLLM, Local LLM, etc

Same principle we described for LLama CPP and Ollama will be applied to other platforms and their LLMs.

CLI (Terminal UI) features

- Testing
- Refactoring
- Planning
- Builder (Code generation) - Creates the code using LLMs. It spwans multiple builders which will split the codebase into modules and generate the code for each module. Each worker (LLM builder) will generate the code for single module. All this will be done in parallel and synchornized. Default number of workers (builders) is one. User can set the number of workers to be used. There will be another builder running - the coordinator which will accept work done by workers and put it properly so no conflicts happen!
- Refactorer - Performs codebase refactoring

- Tester - Executes all available tests, brings up all required apps and emulators as precoditions, generates reports, fixes discovered issues and finall does the SonqrQube and Snyk scanning.
- Debugger
- Diagrams (creation of UML diagrams and others from the Project and its docs and source code in supported formats - drawio, Mermaid.js, png, jpeg, uml, pdf, etc.) Default is drawio.
- Deployment (deploying the project to the cloud or any other defined endpoint)
- Default project is the current directory
- If no Helix.md is available user will be asked to allow creation of it
    - Creation will scan the whole project up to the details and write to the file
    - Format and rules are same as for AGENTS.md or CLAUDE.md
    - If any existing file is found of other agents like CLaude or OpenCode it will be used to support the project's Helix.md generation
    - We will detect all sub-projects and modules. For each of it individual agent file (Helix.md) will be created
- User is able to switch between projects and modules (into deeper hierarchy and to go back)
- User is able to send command sequentially (requests) to the CLI agent which will execute them
- Using tab user can switch between the modes (Builder and others)
- User will be asked about every action that agent is going to do such as write into the file, execute the system command, util, etc. There will be option to skip, accept, accept for the whole session, accept forever for the current project
- All configuration and choices are going to be written in json configuration file: ~/.config/Helix/helix.json
- User will be able to switch modles by choosing provider (LLama CPP, DeepSeek, Qwen, Claude, etc.) and proper model (this will be stored in helix.json file as setting)
- All this operation will be dopne by proper set of commands sent to the agent, we will use a exact same command like Crush has with some differences: we will be able to paste from clipboard request that we want to send; we are going to be able to repeat all requests sent to agent. All requests will be stored in Postgres database using dagtabse component. The whole history of requests will be rememebred so we can scroll through it and repeat any request.
- User will be able to add requests from history to favorites and to group them! There will be command / section so user can pick some and execute regularly! All this is going to be sotred in the database
- Export and import of helix.json and the database will be possible - in sql and json formats
- Theming - Default theme will be based on default ascii code generated LOGO presented at the top of Terminal UI. It will be in green-ish colors. However we will have the following color themes which user can change too: warm red, blue, green, yellow, gold, grey, white, darcula, dark blue, violet, warm orange, etc. User will be able to clone the theme and change the value, save the theme under the new name. All this will be stored in the database and will be loaded when user starts the application.
- Pause / Resume / Detach - User will be able to pause working session and to continue it later! Also, it will be able to detach from it which will not interrupt current session but will allow to continue in the background (non-interactive mode with all questions and permissions accepted).
- Rollback - User will be able to rollback to the previous state of the project! By chosing one of requests that have been completed all work of it will be reverted. User can chose chained rollback - to rollback the request done (changes) and everything that has later been done (other requests) with relation to it, or just rollback the single request work done. No module can stay disabled or broken or the test so in this operation mode all polishing and fixing will be performed to have project in usable stable state after the rollback work has been done.

CLI (Headless) features

- All above but without UI - executed interactively or non-interactively (non-interactive mode is useful for automation and default one)

REST API features

- All above but without UI - executed interactively or non-interactively (non-interactive mode is useful for automation and default one) through REST API

Session work

- All started work will define a session
- Session can have a name
- All active session within a Helix agent will be stored in the database
- It will be possible to join to existing session using proipe "join" command and interacti with it from multiple places - for example user is connected via SSH or REST API from various computers. All this must work.
- It will be possible to kill session which is stuck or we do not want it anymore with command "kill" passed to Helix CLI from the terminal
- It will be possible to list all active sessions with command "sessions"
- Filtering sessions by active, stuck, completed, etc.
- Session monitoring - show commands executed by the agent and their results, cpu and memory usage, etc. ("monitor" command)

Memory

All work done will be stored in database in several layers as working memory so LLMs will be able to use it and continue where they left of! The structire of remembering will be Project -> All done so far -> Current sessions -> Current work in progress. Once user re-executes the request, or types similar one and sends it to execution, system will recognize it as already in progress and will not execute it again! It will update on current status after check and next steps, then continue with work. If there is no such work done on project at the moment fresh new operation will be done to achieve golas from the request sent.

Testing

Testing mode will be specifically used to test modules and project - running all tests, providing it all mandatory dependencies and writing test reports. All discovered issues will be resolved until 100% of tests is executed with succes. Mandatory SonarQube and Snyk testing will be performed.

Project testing

All codebase of Helix CLI must be covered with all test types mentioned in this document. All test types must have coverage of codebase of 100%. Every single test and test case must execute 100% with success! It is mandatory to verify everything with SonarQube intensive deep testing and Snyk! Both will be performed with free versions (docker compose containers). Reports will be generated and any discovered issues will be fixed.

Documentation

For all features, modules, components, tests, everything must be created proper and detialed documentation. We must support end users documentation and developers documentaton. Every single document written has to be available in pure nicely styled too!

Diagrams

In this mode user will be able to create diagrams in proper format for entire project and for specific modules. Default export format is drawio, but user may want to export into a different one or multiple formats at the same time. All this will be supported and will be available in the project.

Helic CLI itself will have allk diagrams generated and ready for users to analyse. We must have them all integrated as the part of the documentation - Mermaid.js io markdowns.

Deployment

In deployment mode user will be able to cretae deploymewnt profiles which will be executed and project to be delivered to the cloud or any other defined endpoint.

Helix CLI will be possible to build and deploy for all operating systems as the multiplatform solution.

Supported deployments and flavors

As we have mentioned we will have default Terminal UI CLI Agent, but pure CLI (Headless) as well. All this to run efficiently on all major operating systems such as: Linux, macOS, BSDs, Windows.

REST API flavor will have several client adaptations so on top of this the following clients will be implemented to cover all features offered by the Terminal UI! Web client, Desktops and Mobile apps. They will all access the local or remote REST API endpoints (which could be local host by default or configurable remote instance).

Automation scripts

All steps - building, testing, generating the documentation, diagrams, etc. will be fully automatized by proper bash scripts.

Launcher icon

Whenever Helix CLI process in some of its forms is running to the user in its operating system proper icon in some form (launcher icon) will be presented. For that purpose generate the icon(s) from the Logo.png from the Assets folder.

Unsorted points

These points have to be assigned as extension to the sections and points from above

- Design - Besides Plan, Build, etc we have another one mode - Design. It creates all designs in Figma and Penpot formats
    - Import Figma created designs into the Figma project via connected Figma account (if any available)
    - Penpot as well

  - Figma and Penpot configuration wizrd as part of every Helix CLI variant - Terminal UI, CLI, REST API, etc.
  - Figma and Penpot configuration will be part of helix.json configuration
- Any design change we do creates new new version which is then imported into Figma project using the API
- Design mode: Import from Figma and Apply to the project
  - Having possibility of syncing changes in both directions
  - Same applies for the Penpot
- Modes flow - Modes can call each other during the development and realisation of the requests (modes may produce various results), each called mode will use its worker LLM model
- Hierarchy if mode calls with workers working in them always presented. Also hierarchi relative to project -> module -> submodule -> modes calls (add this into the architecture)
- Models - one for all modes or different or mixed per mode, dynamic obtained as well (determine which available model is most proper for current mode - Planner, Builder, etc.)
- Create use cases, all flows, all edge cases, documented as real tests to be implemented in e2e and full-automation tests.
- helix cli dockerization with exposed clis - proper documentation
- Predefined mode flows: planning -> building -> testing -> deployment or, create flow dynamically - start the planning mode from which will be decided how the whole implementation flow will look like, then run it
- Log everything into the database
  - Logs viewer (load logs from the whole project, whole module, session, individual request or any step) for analysis and research
  - Support proper filtering
- Clients - Each client will be able to: browse projcts, modules, sessions or indicidual requests and then to join them or manipulate each of it remotely (using REST API)
  - Every running process of Helix CLI will bring up REST API whoch will be able to be used by the clients
  - REST API will bind to default port or port from the helix.json configuration. If it is taken, it will bind to the first next abailable port
  - REST API and clients will support broadcasting of configuration and discovery of service. All client apps will be able to chose between discovered REST API instances or to configure one manually.
- Hardware analysis which is base for selection of proper model and its size will be perfrormed deeply and all detected hardware maximally utilized - vram, ram, cpus, etc.
- Obtain details and capabilities from models and providers dynamically - context size allowed, api detials, etc. Based on this our API will be able to be used for all modes and to send proper instructions - Tooling and Thinkng, etc.
- Problems tollerance handling - To fail if problems arise, or to try to fix them - this is the default setting (settings for this go in helix.json).
- Join by QR code - User will be to join session from other terminal or device by QR code.
- Obtain all models from OpenCode - Zen provider - Free first - Incorporate all free models and other supported from OpenCode Zen (API, etc.)
- Sharing - Access to Project, Session, etc (protected with credentials if accounts available - optional) - We must be able to share whole project, module, session, etc. by link. If accounts available - we

must be able to share by account / credentials protection so that only authorized user will be able to access it.

- Accounts management (optional) - Create account, login, logout, etc. Accounts will have permissions to access projects, modules, sessions, etc. All this has to be configurable. Settings for this will go in Postgres datrabase.
- Rest uses http3 - All REST API will use http3 / quic / cronet and all clients that will communicate with it
- One model for modes - single vs multi instance - Make sure that if we decide to use same model for multiple modes, will it be one same shared instance, or instance per mode - this will be configurable.
- Porting mode - Mode for porting applications from one codebase into others. Identifiying replacements for destination technology and creating equivalent to the source one.
- Fallback models - If some model fails with the execution (API fails), use first configured fallback model (if any). All this will be configured in helix.json and proper UI / UX defined for it.
- Models configuration wizard for project, module, etc. - Models configuration will allow to define models used by different modes or hierarchy levels - general global, per project, per module, per sesssion, etc - this will be one part sotred in helix.json and another part in database (per hierarchy instance item such as individual project, individual module, individual session). All global level goes to helix.json.
- Multiple api keys support - For each model which requires API key we will have to support multiple keys. Then, when it comes to multiple instances of the model use we may always use the first available key.
- Models processes manager / monitor with performance monitoring and alerts - Entering this part of Terminal UI or one of the client's UIs for this we will see all active models, where are they used and how much resources they are consuming. If system resources are draind proper warning or errors have to be presented. If we hit critical moment - system starts to glitch - running sessions have to be paused!
- Delegating work to multiple machines - nodes (slaves) - instead of join command to have connect or similar. We must be able to do the work with multiple computers which are all joined to the same session. Or multiple sessions done by muultiple machines in parallel for one project, or module, etc. Like this each instance can instantiate the model and do the work. BAsed on exisiting mechanism it will run models with maximal performance and capabilities.
- Remote share connect - We must be able to request connect from other machines (previous point extended), or to send request to remote machines to connect. For this has to be proper mechanism. Will machines automatically accpet the requests to connect and give its resources for joined work will be defined in helix.json as the part of the configuration. It will be configurable from all clients, cli and terminal UI.
- For all communication between machines running Helix CLI we will use REST API. It will be configurable in helix.json as the part of the configuration. It will be configurable from all clients, cli and terminal UI. For all events that occur proper websocket events will be sent in real time and all connected participants will act on it.