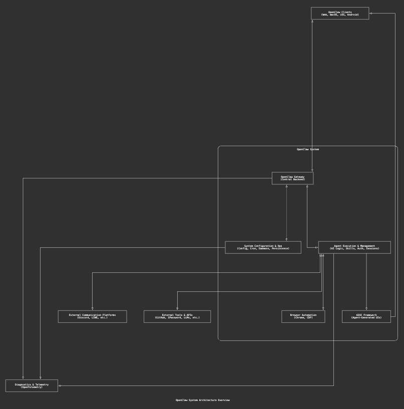
openclaw/openclaw


OpenClaw Clients
(Web, macOS, iOS, Android)
OpenClaw System
OpenClaw Gateway
(Central Backend)
System Configuration & Ops
(Config, Cron, Daemons, Persistence)
Agent Execution & Management
(AI Logic, Skills, Auth, Sessions)
External Communication Platforms
(Discord, LINE, etc.)
External Tools & APIs
(GitHub, 1Password, LLMs, etc.)
Browser Automation
(Chrome, CDP)
A2UI Framework
(Agent-Generated UIs)
Diagnostics & Telemetry
(OpenTelemetry)
OpenClaw System Architecture Overview

OpenClaw is a software system for deploying and orchestrating AI agents that automate complex tasks across digital environments and communicate through various channels. It enables agents to interact with web browsers, manage diverse messaging platforms, execute shell commands, and integrate with a wide array of external services. The system also offers robust client applications and a flexible user interface for agent-driven interactions.

OpenClaw's agent functionality covers communication, browser automation, and integration with various communication channels. The Agent Client Protocol (ACP) facilitates agent interaction with a backend gateway, supporting command-line sessions and session management. Agent Client Protocol (ACP) Implementation describes this in detail. Agents can control Chrome-based browsers using the Chrome DevTools Protocol (CDP), managing their lifecycle, profiles, and interactions such as clicks, typing, and navigation. Further information is available in Browser Automation and Control. A plugin architecture integrates diverse messaging platforms, handling message processing, authorization, and channel-specific configurations. See Communication Channel Management for more information.

Agents leverage an extensive set of skills to interact with external tools and services, enabling actions such as secure secret management, note-taking, content monitoring, and home automation. A dedicated system manages the installation, authentication, and configuration of these integrations. The macOS environment includes a wake-word hook daemon, which provides local voice transcription and shell command dispatch, allowing agents to respond to voice commands. Agent Skills and Tooling provides an overview of these capabilities.

User interaction occurs through client applications and dynamic user interfaces. OpenClaw provides native client applications for Android, iOS, and macOS, offering functionalities such as Canvas, Chat, and Camera. Client Applications covers their architecture and features. The A2UI (Agent-to-User Interface) framework enables AI agents to generate rich, updateable user interfaces using a declarative JSON format. This framework prioritizes security, LLM-friendliness, and framework-agnostic portability, allowing clients to render agent-described UI components dynamically. More details are in A2UI Framework for Agent-Generated UIs.

The system's modularity is enhanced through extensions and integrations. This includes plugins for authentication providers, agent tools for advanced tasks like LLM-based JSON output generation, structured workflow execution, and long-term memory management with vector search. A Chrome extension facilitates browser automation by relaying CDP commands. OpenClaw Extensions and Integrations explores these aspects.

Operational components include a robust command-line interface (CLI) for managing the system, a cron job management system for scheduling automated tasks, and daemon services for background operations across different operating systems. Application configuration is centrally managed, ensuring consistent settings across components. A dedicated development environment, `Clawdbot`, supports workflows for synchronization, component rebuilding, and troubleshooting. These infrastructure aspects are detailed in Core Agent Development and Workflow. Development operations are further supported by various utility scripts for building, testing, monitoring, and documentation internationalization.

Core Agent Development and Workflow

The primary source code for the OpenClaw project, located in `src`, governs agent communication, browser automation, channel management, command-line interface (CLI) utilities, application configuration, cron job scheduling, and daemon services. It defines how agents interact, how external systems are controlled, and how the OpenClaw platform itself operates.

Agent communication is facilitated by the Agent Client Protocol (ACP), which provides both client-side and server-side capabilities for interaction with a backend gateway. The client-side, implemented in `src/acp/client.ts`, allows for interactive command-line sessions and manages the child process running the ACP server. The server-side, primarily in `src/acp/server.ts`, translates ACP requests into gateway commands, enabling backend interactions and comprehensive session management. Agent Client Protocol (ACP) Implementation further details this protocol.

The management of OpenClaw agents is handled by the code in `src/agents`. This includes resolving agent directory paths, applying file system patches to modify agent configurations, and managing authentication profiles. Authentication profiles, located in `src/agents/auth-profiles`, support API key resolution and OAuth token refreshing, providing health summaries for authentication statuses. This area also includes the core logic for executing bash commands with approval mechanisms and process management. For more information, see Agent Management and Authentication.

Automated message processing and agent replies are orchestrated through the code in `src/auto-reply`. This includes functionalities such as text chunking for efficient message handling, command authorization and detection, and the dispatching of inbound messages. It also supports the generation of media notes and the extraction of model directives from messages, culminating in the orchestration of AI agent replies, including session state management, memory flushing, and streaming capabilities. Details are available in Automated Message Processing and Replies.

Browser automation and control are managed by the code in `src/browser`. This encompasses the entire lifecycle of Chrome-based browsers, from launching and termination to configuration resolution and profile management. Interactions with the browser are primarily achieved through the Chrome DevTools Protocol (CDP), allowing for detailed control and observation. See Browser Automation and Control for further information.

Web canvas applications are hosted and served by components in `src/canvas-host`. This includes the hosting of `a2ui` framework applications, static file serving, live-reloading during development, and secure file access. More information can be found in Web Canvas Applications Hosting.

Utilities and a plugin architecture for managing various communication channels are found in `src/channels`. This includes features like acknowledgement reactions, allowlisting, channel configuration, chat type normalization, command gating, and conversation labeling. Communication Channel Management provides a more detailed overview.

The OpenClaw command-line interface (CLI) is implemented within `src/cli`. This includes robust argument parsing, dynamic shell completion, and specialized commands for ACP interactions, browser automation, channel account management, and configuration management. For a comprehensive look, refer to Command-Line Interface (CLI) Utilities.

The definition and implementation of all agent-related commands reside in `src/commands`. This includes the orchestration of agent message processing, session management, model selection, skill integration, and result delivery, with a focus on configuration utilities and provider bindings. Refer to Agent Command Execution for more details.

Application configuration for OpenClaw is centrally managed in `src/config`. This covers input/output operations, persistence, validation against a schema, migration of legacy configurations, default settings, environment variable substitution, and path management. Additional information is available in Application Configuration Management.

A comprehensive cron job management system is implemented in `src/cron`. This system handles job definition, normalization, scheduling, execution with isolated agent turns, service management for job lifecycles, persistence, and run logging. For more details, see Cron Job Management System.

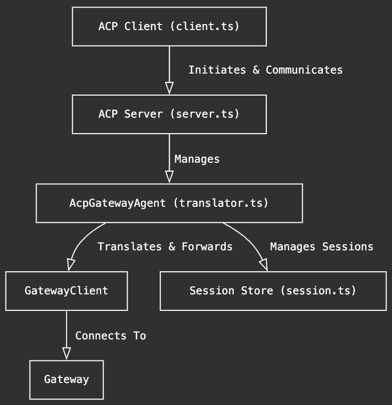
Daemon services, specifically the gateway and node services, are managed across different operating systems by the code in `src/daemon`. This includes their lifecycle management, platform-specific implementations (macOS, Linux, Windows), path resolution, program argument determination, and environment variable construction. Daemon Service Management provides a deeper dive.

The `Clawdbot` development environment, described by files in `.agent`, outlines workflows for synchronizing local forks with upstream repositories, rebuilding components, verifying functionality, and troubleshooting macOS and Swift compatibility issues. These processes are essential for maintaining the integrity and functionality of the development setup. For more information, see Clawdbot Development Environment Management.

Extensions for a coding agent, located in `.pi`, enhance functionality through Git integration, file interaction tracking, GitHub URL processing, and Text User Interface (TUI) statistics. This directory also contains structured prompts that guide software development workflows, including changelog auditing, GitHub issue analysis, and pull request management. Coding Agent Extensions and Prompts details these capabilities.

Legacy package compatibility is addressed in `packages`. This part of the codebase manages the transition from older package names like `clawdbot` and `moltbot` to the current `openclaw` package by re-exporting APIs and issuing post-installation warnings to users about the name changes.

Agent Client Protocol (ACP) Implementation



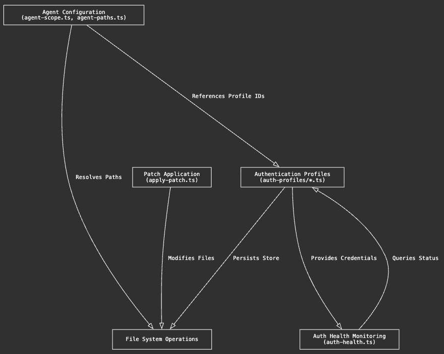
The Agent Client Protocol (ACP) provides a standardized method for `openclaw` to communicate with backend services, acting as a bridge between interactive client applications and a gateway. This protocol facilitates the agent's ability to process commands, manage ongoing sessions, and interact with various backend functionalities, particularly in environments like integrated development environments (IDEs).

The client-side implementation, primarily defined in `src/acp/client.ts`, allows for interactive command-line sessions. This client can spawn an `openclaw` child process to serve as the ACP server, managing its lifecycle and facilitating communication over NDJSON streams. It handles session notifications, such as agent messages and tool calls, and provides a command-line interface for users to send prompts and receive responses. A notable feature is its automatic handling of permission requests from the agent, simplifying the interactive experience.

On the server side, `src/acp/server.ts` establishes the ACP server that integrates with a backend gateway. This server loads configuration, resolves gateway authentication details, and initializes a `GatewayClient` to manage the connection to the backend. The `AcpGatewayAgent`, detailed in `src/acp/translator.ts`, acts as a central component, translating ACP requests into gateway-specific commands and processing events from the `GatewayClient` to update the ACP client. It manages session lifecycle, handles user prompts, and processes tool calls, ensuring smooth interaction between the client and the gateway.

A crucial aspect of the ACP implementation is its comprehensive session management, which is handled by modules like `src/acp/session.ts` and `src/acp/session-mapper.ts`. The system provides an in-memory store for `AcpSession` objects, allowing for the creation, retrieval, and tracking of sessions. It also supports associating sessions with active runs and managing their cancellation through `AbortController`. The `session-mapper` is responsible for parsing session metadata from incoming requests, resolving session keys based on labels or explicit keys, and conditionally resetting sessions via the `GatewayClient`. This ensures that sessions are correctly identified, maintained, and cleaned up as needed, supporting dynamic and persistent interactions. The available commands that an agent can understand and process are defined in `src/acp/commands.ts`, providing a central registry for supported functionalities.

Agent Management and Authentication



OpenClaw agents require structured management for their operational integrity, encompassing their location on the filesystem, their behavioral configurations, the application of changes to their local environment, and robust authentication mechanisms to interact with external services.

The system determines the agent's root directory by first checking specific environment variables such as `OPENCLAW_AGENT_DIR` or `PI_CODING_AGENT_DIR`. If these are not defined, it defaults to a path within the OpenClaw state directory. This process ensures that agents can locate their resources consistently across different deployment environments. The function `resolveOpenClawAgentDir` in `src/agents/agent-paths.ts` is responsible for this resolution, with `ensureOpenClawAgentEnv` ensuring environment variables are set accordingly.

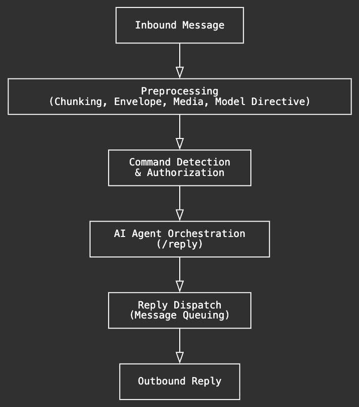
Agent configurations define an agent's operational parameters, including its name, workspace, assigned model, identity, and available tools. The system can list agents, identify the default agent, and resolve session-specific agent IDs. The primary model and any fallback models for an agent are also determined from these configurations. Functions such as `resolveAgentConfig`, `resolveAgentModelPrimary`, and `resolveAgentModelFallbacksOverride` in `src/agents/agent-scope.ts` facilitate this configuration management. Workspace and agent-specific directories are resolved to ensure proper isolation and access control.

To enable agents to modify their local environment, the system includes a file system patch application mechanism. This allows agents to perform operations like adding, deleting, updating, or moving files based on a structured patch format. The `apply_patch` `AgentTool`, created by `createApplyPatchTool` in `src/agents/apply-patch.ts`, orchestrates these changes. The `applyPatch` function interprets the patch input, validates paths to ensure operations occur within specified working or sandbox directories, and executes the file system modifications. This includes parsing the patch text into individual change "hunks," each describing a file operation. For file content updates, `applyUpdateHunk` in `src/agents/apply-patch-update.ts` applies line-based replacements, with `seekSequence` providing robust pattern matching to handle minor formatting differences.

Authentication is managed through a comprehensive profile system that supports various credential types, including API keys and OAuth tokens. This system handles loading, saving, and migrating authentication profiles, as well as merging profiles from different sources (e.g., a main agent and a specific agent). A critical feature is the ability to refresh OAuth tokens automatically when they expire. The functions within the `src/agents/auth-profiles` directory are central to this, particularly `resolveApiKeyForProfile` in `src/agents/auth-profiles/oauth.ts`, which manages token refreshing and ensures continuity of access to external services. The system also tracks profile usage, applying cooldowns for failed authentication attempts to prevent abuse or further issues.

The health of these authentication profiles is continuously monitored. The `buildAuthHealthSummary` function in `src/agents/auth-health.ts` assesses the status of each profile and its associated provider, classifying them as "ok," "expiring," "expired," "missing," or "static." For OAuth credentials, the presence of a refresh token determines if an expired access token can still be considered "ok" due to its auto-renewable nature. This provides an overview of the authentication system's operational status, identifying potential issues before they impact agent functionality.

Automated Message Processing and Replies


Inbound Message
Preprocessing (Chunking, Envelope, Media, Model Directive)
Command Detection & Authorization
AI Agent Orchestration (/reply)
Reply Dispatch (Message Queuing)
Outbound Reply

The `auto-reply` module at `src/auto-reply` contains the core logic for processing inbound messages, authorizing commands, and orchestrating AI agent replies within the OpenClaw system. It manages the lifecycle of agent interactions, from initial message reception to the generation and delivery of responses.

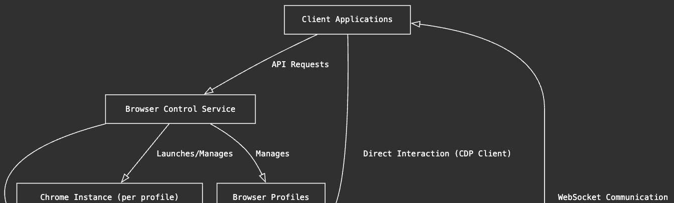
Message processing begins with utilities for chunking and formatting text. For outbound messages, text is split into manageable chunks based on configurable limits and modes (e.g., by length, newlines, or paragraphs) using functions such as `chunkText` and `chunkMarkdownText` within `src/auto-reply/chunk.ts`. This ensures messages adhere to platform-specific size constraints while preserving content integrity, especially for Markdown-formatted text. Message metadata, such as timestamps and sender information, is formatted into "envelopes" for consistent display and logging, as handled by functions like `formatAgentEnvelope` and `formatInboundEnvelope` in `src/auto-reply/envelope.ts`.

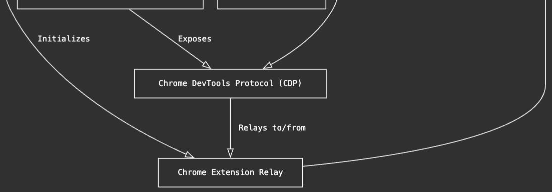
Command management and authorization are central to the system's security and functionality. The module includes logic for detecting control commands and inline command tokens within message text using functions like `hasControlCommand` and `hasInlineCommandTokens` in `src/auto-reply/command-detection.ts`. Before executing a command, the system determines if the sender is authorized using `resolveCommandAuthorization` in `src/auto-reply/command-auth.ts`, which resolves sender identities, checks against owner lists, and considers channel-specific configurations. Commands themselves are defined, registered, and parsed via a comprehensive registry, with utilities for handling aliases and argument parsing found in `src/auto-reply/commands-registry.data.ts` and functions like `parseCommandArgs` and `serializeCommandArgs` in `src/auto-reply/commands-registry.ts`. This allows for a flexible and extensible command-line interface for agents.

Inbound message dispatching is orchestrated by `dispatchInboundMessage` in `src/auto-reply/dispatch.ts`, which processes incoming messages based on system configuration. This includes utilities for debouncing and deduplicating messages to prevent redundant processing. The system also supports the generation of media notes to describe attached media, as implemented by `buildInboundMediaNote` in `src/auto-reply/media-note.ts`. Model directives and optional aliases are extracted from messages using `extractModelDirective` in `src/auto-reply/model.ts`, allowing agents to dynamically select or configure AI models for their responses.

The orchestration of AI agent replies is a key function, primarily managed by the `src/auto-reply/reply` subdirectory. This includes running agent turns, flushing memory as needed, and processing reply payloads. The system also handles session aborts and manages typing signals to provide a more natural interaction experience. Interactive execution of bash commands, when authorized, is also integrated into the reply orchestration. Furthermore, the system includes logic for managing group activation settings, allowing the system to determine whether to activate on explicit mentions or always, as detailed by `parseActivationCommand` and `normalizeGroupActivation` in `src/auto-reply/group-activation.ts`. Heartbeat-related logic, including default prompts, timings, and the ability to detect effectively empty heartbeat content to optimize API calls, is handled by functions like `stripHeartbeatToken` and `isHeartbeatContentEffectivelyEmpty` in `src/auto-reply/heartbeat.ts`.

Browser Automation and Control





The OpenClaw project includes a robust system for managing and interacting with Chrome-based browsers, which is crucial for automation tasks. This system handles the complete lifecycle of browser instances, from launching and configuration to termination, and facilitates detailed interactions through the Chrome DevTools Protocol (CDP).

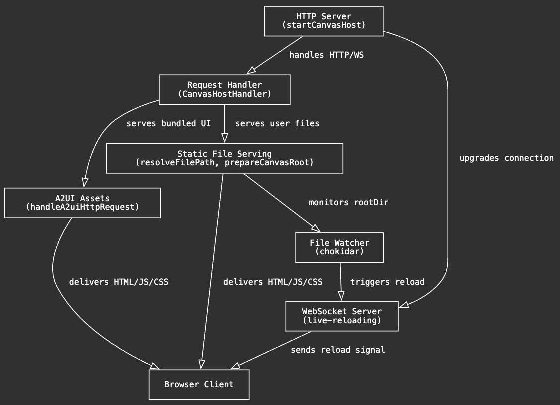
Central to this capability is the management of browser instances, defined by `RunningChrome` in `src/browser/chrome.ts`, which tracks the process ID, executable details, user data directory, and CDP port of a running browser. The system leverages `launchOpenClawChrome` in `src/browser/chrome.ts` to launch a browser, employing a comprehensive sequence that includes ensuring port availability, resolving the browser executable via `resolveBrowserExecutableForPlatform` in `src/browser/chrome.executables.ts`, and setting up a dedicated user data directory with `resolveOpenClawUserDataDir` in `src/browser/chrome.ts`. This directory ensures profile isolation, preventing conflicts between different browser sessions. The launch process also incorporates steps for bootstrapping a profile, decorating it with OpenClaw-specific settings using `decorateOpenClawProfile` in `src/browser/chrome.profile-decoration.ts`, and ensuring a clean exit state through `ensureProfileCleanExit` in `src/browser/chrome.profile-decoration.ts` to avoid crash recovery prompts.

Configuration resolution is handled by `resolveBrowserConfig` in `src/browser/config.ts`, which parses and validates browser settings, including executable paths, ports, profiles, and headless modes. This function ensures that default values are applied, and custom settings are correctly integrated. Profile management is supported with capabilities to create, delete, and list browser profiles, each with its own data directory, CDP port, and potentially custom appearance (color and name). The `decorateOpenClawProfile` function in `src/browser/chrome.profile-decoration.ts` applies these settings to a profile's configuration files.

Interactions with the browser are primarily facilitated through the Chrome DevTools Protocol (CDP). Utilities in `src/browser/cdp.ts` provide functions for high-level operations such as `captureScreenshotPng` for capturing screenshots, `evaluateJavaScript` for executing arbitrary code within the browser context, and `createTargetViaCdp` for creating new browser tabs or pages. Lower-level CDP communication, including WebSocket connections and authentication handling, is managed by helper functions such as `createCdpSender` and `withCdpSocket` in `src/browser/cdp.helpers.ts`. The system also normalizes CDP WebSocket URLs using `normalizeCdpWsUrl` in `src/browser/cdp.ts` to ensure consistent communication.

For external interaction, a client API is available, providing functions to manage browser status, profiles, and tabs. The `browserStatus`, `browserProfiles`, and `browserOpenTab` functions in `src/browser/client.ts` facilitate these operations. A central `browserAct` function in `src/browser/client-actions-core.ts` handles various UI interactions, including clicks, typing, and navigation, using a type-safe discriminated union (`BrowserActRequest`) to specify actions. Observational functions like `browserConsoleMessages` and `browserRequests` in `src/browser/client-actions-observe.ts` allow monitoring browser events, while `browserCookiesSet` and `browserSetOffline` in `src/browser/client-actions-state.ts` enable control over browser state. All client-side requests are routed through `fetchBrowserJson` in `src/browser/client-fetch.ts`, which ensures consistent communication with the browser control service and enhances error reporting. An `extension-relay` in `src/browser/extension-relay.ts` also allows external Chrome extensions to interact with the CDP via a local server. For more on the Chrome extension, see Chrome Extension for OpenClaw Gateway.

Web Canvas Applications Hosting



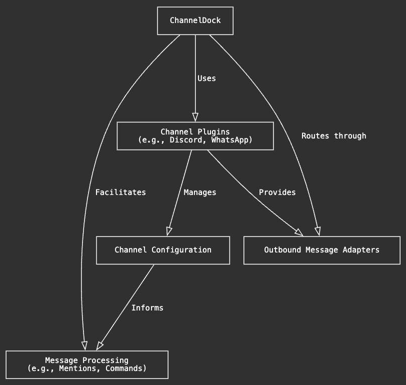
The OpenClaw project hosts and serves web canvas applications, with a particular focus on integrating the `a2ui` framework. This functionality is primarily managed within the `src/canvas-host` directory. The system provides a robust mechanism for serving static files, enables live-reloading during development, and incorporates security features to prevent unauthorized file access.

The core functionality is exposed through two main interfaces: `createCanvasHostHandler` and `startCanvasHost`. `createCanvasHostHandler` establishes a handler responsible for processing HTTP requests and WebSocket upgrades. This handler dynamically determines the application's root directory, ensures the presence of an `index.html` file (creating a default one if needed), and, when enabled, monitors the root directory for file changes to trigger live reloads via WebSockets. The system securely serves static files by normalizing URL paths and verifying that requested files reside within the designated root directory, thereby mitigating directory traversal vulnerabilities. HTML files are dynamically modified to inject a client-side live-reload script and communication bridges for interacting with native applications.

`startCanvasHost` initializes and launches an HTTP server that leverages a `CanvasHostHandler`. This server handles incoming requests, prioritizing those specifically for `a2ui` assets, before delegating to the general canvas handler for other static content. It also manages WebSocket upgrade requests for the live-reload feature.

The integration with `a2ui` is a key aspect of this hosting service. The `src/canvas-host/a2ui/index.html` file provides the visual environment for `a2ui` applications, featuring dynamic background animations, a canvas element for rendering, platform-specific theming (especially for Android), and a debug status overlay. This page exposes a global `window.__openclaw` object, allowing external scripts to interact with the canvas and status display. The `src/canvas-host/a2ui.ts` file manages the serving of `a2ui` assets, performing secure path resolution and dynamically injecting the live-reload script into HTML responses. This dynamic injection facilitates cross-platform communication between the web view and native code, ensuring seamless development workflows and a responsive user experience. Further details on the `a2ui` framework can be found in A2UI Framework for Agent-Generated UIs.

Communication Channel Management


ChannelDock
Uses
Routes through
Channel Plugins (e.g., Discord, WhatsApp)
Facilitates
Manages
Provides
Channel Configuration
Outbound Message Adapters
Informs
Message Processing (e.g., Mentions, Commands)

The OpenClaw project incorporates a comprehensive system for managing interactions across diverse communication platforms. This system is built around a flexible plugin architecture and a set of core utilities designed to standardize channel-specific behaviors.

Central to channel management is the concept of a `ChannelDock`, a lightweight representation of a chat channel that provides a consistent interface for accessing its metadata and capabilities. This allows the system to interact uniformly with different channels, whether they are built-in or dynamically loaded via plugins. The `DOCKS` constant in `src/channels/dock.ts` serves as a central registry for these built-in channel configurations, while the `buildDockFromPlugin` function in the same file enables dynamic integration of plugin-defined channels.

Channel configuration is managed through utilities that handle matching entries, normalizing channel identifiers (or "slugs"), and resolving configurations with fallback mechanisms. For example, `normalizeChannelSlug` in `src/channels/channel-config.ts` standardizes channel names into a consistent format. Access control for channel configurations, including the ability to write settings, is determined by functions such as `resolveChannelConfigWrites` in `src/channels/plugins/config-writes.ts`, which respects both account-specific and channel-wide settings.

The system includes robust logic for processing messages, such as normalizing chat types. The `normalizeChatType` function in `src/channels/chat-type.ts` converts various input strings (like "dm" or "group") into a standardized `NormalizedChatType` (e.g., "direct" or "group").

Command gating is another key feature, ensuring that commands are only executed when properly authorized. The `resolveCommandAuthorizedFromAuthorizers` function in `src/channels/command-gating.ts` determines if a command is authorized based on access group usage and a list of authorizers. Additionally, `resolveControlCommandGate` provides a mechanism to evaluate if a command should be blocked, particularly for text commands when access groups are not active.

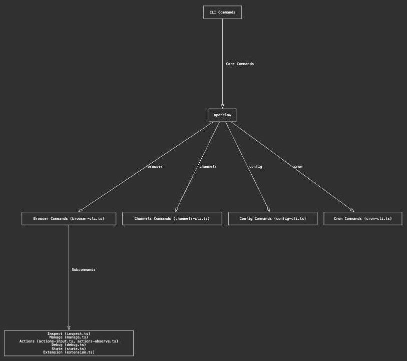
Automated message reactions, such as acknowledgment reactions, are managed by logic that determines when reactions should be sent and, in some cases, removed after a reply. Functions like `shouldAckReaction` in `src/channels/ack-reactions.ts` evaluate conditions like the message scope and type to decide if a reaction is appropriate.

To provide user-friendly references, conversation labeling utilities are available. `resolveConversationLabel` in `src/channels/conversation-label.ts` generates a human-readable label for a conversation by prioritizing explicit labels, thread labels, sender names for direct chats, and group subjects for group or channel conversations.

Mention gating helps manage message processing based on whether an agent is mentioned. The `resolveMentionGating` and `resolveMentionGatingWithBypass` functions in `src/channels/mention-gating.ts` determine if a message should be skipped based on mention requirements and detection capabilities, including conditions that allow bypassing strict mention rules for certain authorized commands.

The plugin architecture, primarily located in `src/channels/plugins`, enables seamless integration of various communication platforms like Discord, Signal, Telegram, and WhatsApp. It supports discovery, loading, and caching of channel plugins, as well as onboarding processes through `ChannelOnboardingAdapter` implementations, which guide the setup and configuration of new channels. This architecture also supports channel-specific message actions, outbound messaging capabilities, and directory services for extracting and normalizing contact information across platforms.

Command-Line Interface (CLI) Utilities


CLI Commands
Core Commands
openclaw
browser
channels
config
cron
Browser Commands (browser-cli.ts)
Channels Commands (channels-cli.ts)
Config Commands (config-cli.ts)
Cron Commands (cron-cli.ts)
Subcommands
Inspect (inspect.ts)
Manage (manage.ts)
Actions (actions-input.ts, actions-observe.ts)
Debug (debug.ts)
State (state.ts)
Extension (extension.ts)

The OpenClaw project features a robust Command-Line Interface (CLI) that serves as the primary interaction point for managing various aspects of the system. This CLI is built upon the `commander` library and provides functionalities ranging from agent communication and browser automation to channel management and configuration. The design emphasizes consistent argument parsing, dynamic shell completion, and clear output formatting across different commands.

Central to the CLI's operation is its argument parsing and handling, which uses utilities in `src/cli/argv.ts` to process command-line inputs. These utilities are responsible for detecting flags, extracting values, and identifying command paths, ensuring that the CLI can interpret user commands accurately regardless of the execution environment. The `buildParseArgv` function within this module normalizes the raw arguments, making them consistent for subsequent processing. For user-friendly output, the CLI includes a banner display mechanism in `src/cli/banner.ts` that provides version information and project branding while avoiding redundant emissions and adapting to terminal capabilities.

The CLI integrates with the Agent Communication Protocol (ACP) through dedicated commands defined in `src/cli/acp-cli.ts`. These commands allow users to run an ACP gateway server, facilitating communication with OpenClaw's backend services, or to initiate an interactive ACP client session. This enables direct interaction with agents and their capabilities from the command line.

A significant portion of the CLI is dedicated to browser automation and control, managed through the `src/cli/browser-cli.ts` module. This module registers a comprehensive `browser` command with numerous subcommands. These include:

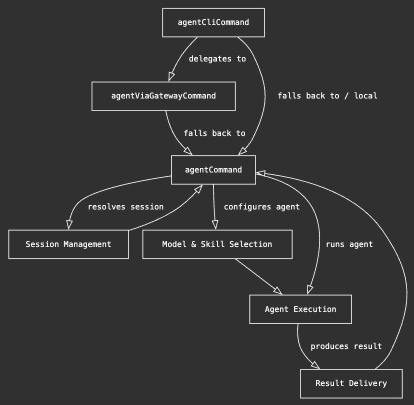
- **Browser Instance Management:** Commands to start, stop, and query the status of browser instances, as well as manage browser profiles and individual tabs. These are registered via `src/cli/browser-cli-manage.ts`.
- **Browser Interaction:** Subcommands for simulating user inputs such as clicks, typing, navigation, and file uploads, handled by the components in `src/cli/browser-cli-actions-input`.
- **Browser Observation:** Commands for retrieving console messages, capturing screenshots, generating page PDFs, and inspecting network response bodies, defined in `src/cli/browser-cli-actions-observe.ts` and `src/cli/browser-cli-inspect.ts`.
- **Debugging and State Management:** Utilities for debugging browser interactions (e.g., element highlighting, trace recording) in `src/cli/browser-cli-debug.ts`, and for setting browser environmental states like viewport, geolocation, or headers in `src/cli/browser-cli-state.ts`. The CLI also supports managing cookies and local/session storage through `src/cli/browser-cli-state.cookies-storage.ts`.
- **Chrome Extension Management:** Commands for installing and resolving the path of the OpenClaw Chrome extension, which facilitates advanced browser control, as detailed in `src/cli/browser-cli-extension.ts`.

For managing communication channels, the CLI provides a `channels` subcommand, registered by `src/cli/channels-cli.ts`. This allows users to list, add, remove, and authenticate various chat channel accounts, including handling authentication flows.

Configuration management is another core function of the CLI, implemented through `src/cli/config-cli.ts`. This module enables users to `get`, `set`, and `unset` application configuration values, supporting path-based access and integration with a configuration wizard for guided setup.

Finally, the CLI includes a cron job management system, with commands registered in `src/cli/cron-cli/register.ts`. This system allows users to define, schedule, edit, enable, disable, and monitor recurring tasks within the OpenClaw environment. For more information on cron jobs, see Cron Job Management System. The CLI also supports dynamic shell completion for various shells (zsh, bash, powershell, fish), enhancing usability by providing context-aware suggestions for commands and arguments through `src/cli/completion-cli.ts`.

Agent Command Execution



Agent-related commands orchestrate the processing of messages, manage sessions, select models, integrate skills, and deliver results. This system supports both gateway and embedded agent interactions, with a focus on configuration utilities and provider bindings.

The primary entry point for agent execution is handled by the `agentCommand` function in `src/commands/agent.ts`. This function coordinates the entire workflow, from validating inputs and resolving agent IDs to managing agent workspaces and loading configurations. It determines the agent's thinking and verbosity levels, handles timeouts, and orchestrates session management, including persistence of settings and updates to the session store after an agent run. The `agentCommand` function also integrates skill snapshots, ensuring agents have access to their defined capabilities, and manages model selection, including fallback strategies if a primary model is unavailable or unsuitable. Execution is delegated to either a CLI agent via `runCliAgent` or an embedded agent via `runEmbeddedPiAgent`, based on the resolved provider. Finally, the system delivers the agent's results using `deliverAgentCommandResult`.

Agent command execution can occur either directly via an embedded agent or through a gateway service, as defined in `src/commands/agent-via-gateway.ts`. The `agentCliCommand` function acts as an entry point, determining whether to use `agentViaGatewayCommand` for gateway interaction or `agentCommand` for local execution, with a fallback mechanism if the gateway fails. This approach provides flexibility and resilience in agent deployment.

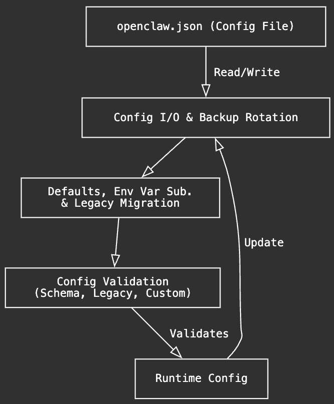
Various sub-functionalities support the core agent command execution. Agent command types and configuration are defined in `src/commands/agent/types.ts`, which specifies structures for multimodal content and command options. Agent run context resolution is managed by `resolveAgentRunContext` in `src/commands/agent/run-context.ts`, which normalizes configuration options into a unified context. Session management, including resolution, loading, and freshness evaluation, is handled by `src/commands/agent/session.ts`, with session store updates (e.g., token usage, model, and provider information) managed by `updateSessionStoreAfterAgentRun` in `src/commands/agent/session-store.ts`. Result delivery, including the orchestration of outbound payloads, is managed by `deliverAgentCommandResult` in `src/commands/agent/delivery.ts`.

Agent management commands provide functionalities for adding, deleting, and managing agent identities. The `agentsAddCommand` in `src/commands/agents.commands.add.ts` facilitates adding new agent configurations, supporting both interactive wizard-guided flows and non-interactive command-line modes. It handles agent ID normalization, workspace setup, model and authentication configuration, and channel binding. The `agentsDeleteCommand` in `src/commands/agents.commands.delete.ts` handles the deletion of agents and their associated data, including configuration entries and file system resources, with user confirmation for safety. Agent identities, including names, emojis, themes, and avatars, are managed by `agentsSetIdentityCommand` in `src/commands/agents.commands.identity.ts`, which can load identity data from `IDENTITY.md` files or command-line options. The `agentsListCommand` in `src/commands/agents.commands.list.ts` generates and displays a list of configured agents, including their identities, workspaces, models, routing rules, and associated providers.

Configuration utilities in `src/commands/agents.config.ts` manage agent configurations, including listing, finding, resolving details, building summaries, applying updates, and pruning agent-related settings. Agent-channel bindings are managed by `src/commands/agents.bindings.ts`, which provides utilities to describe, apply, build, and parse agent bindings, handling conflicts and default account resolution. Communication channel providers and their accounts are managed by `src/commands/agents.providers.ts`, which includes functionalities for listing provider statuses and associating them with agents.

Authentication for various AI providers is handled through a system of authentication choices. The `src/commands/auth-choice-options.ts` file defines and structures authentication choices for user presentation. The `promptAuthChoiceGrouped` function in `src/commands/auth-choice-prompt.ts` facilitates interactive selection of authentication methods. Specific files like `src/commands/auth-choice.apply.anthropic.ts`, `src/commands/auth-choice.apply.api-providers.ts`, and others manage the application of authentication methods for different providers, including API key handling and integration with external authentication plugins.

Application Configuration Management


openclaw.json (Config File)
Read/Write
Config I/O & Backup Rotation
Defaults, Env Var Sub. & Legacy Migration
Config Validation (Schema, Legacy, Custom)
Update
Validates
Runtime Config

OpenClaw's application configuration is centrally managed, providing a robust system for handling application settings. This system is responsible for reading configuration data, ensuring its validity, adapting older configurations to current formats, setting sensible defaults, integrating environment variables, and managing configuration paths.

Configuration data is read from and written to `openclaw.json` files, with mechanisms in place for `backup rotation` to maintain historical versions of the configuration. The file `src/config/config.backup-rotation.test.ts` demonstrates how `writeConfigFile` manages this process, ensuring data integrity and providing recovery points.

A critical aspect of configuration management is validation, which uses a defined schema to ensure that all settings conform to expected structures and values. This validation process includes checks for issues such as invalid broadcast strategies and non-array broadcast entries, as seen when `validateConfigObject` is used in `src/config/config.broadcast.test.ts`. Validation also covers improper gateway remote transport configurations, as confirmed in `src/config/config.gateway-remote-transport.test.ts`, and `identity avatar` paths, which are rigorously checked to be within the agent's workspace or to be valid URLs/data URIs, rather than external or malicious paths, as outlined in `src/config/config.identity-avatar.test.ts`. The system also performs strict validation to reject unknown fields and identify legacy configuration entries without automatically migrating them, allowing for deliberate updates, as tested in `src/config/config.preservation-on-validation-failure.test.ts`.

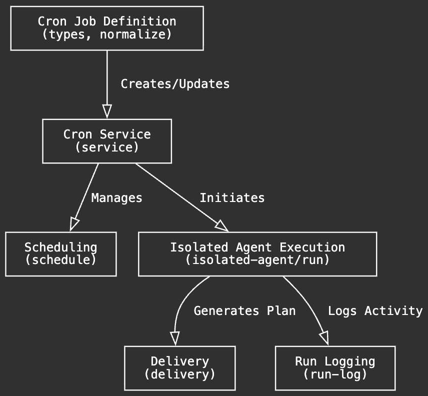
To ensure backward compatibility and smooth transitions, the configuration system includes legacy configuration migration. This process transforms older configuration formats into the current standard, handling changes such as the relocation of routing properties to channel-specific or tool-specific sections, and updating agent and tools properties to their modern counterparts. This migration also ensures that channel-specific `dmPolicy` and `allowFrom` settings are properly enforced. These migration capabilities are extensively tested, as shown in `src/config/config.legacy-config-detection.rejects-routing-allowfrom.test.ts`.

Default settings are applied where configurations are not explicitly provided by the user. This includes default values for `resolveAgentMaxConcurrent` and `resolveSubagentMaxConcurrent`, ensuring operational stability even with minimal user input, which is tested in `src/config/config.agent-concurrency-defaults.test.ts`. For compaction settings, the system defaults the `compaction.mode` to `"safeguard"` when not specified, as verified in `src/config/config.compaction-settings.test.ts`. Identity-based default behavior ensures that properties like `mentionPatterns` and `responsePrefix` are not automatically derived if an explicit identity is set, and `ackReactionScope` defaults to `"group-mentions"`, ensuring that explicit configurations are always respected over derived defaults, as seen in `src/config/config.identity-defaults.test.ts`.

Environment variable substitution allows configuration values to be dynamically sourced from the operating environment. This includes capabilities to apply missing environment variables from the `openclaw.json` configuration and a strict policy of not overriding existing environment variables, giving precedence to system-level settings, as confirmed in `src/config/config.env-vars.test.ts`.

Finally, comprehensive path management is provided to safely access and manipulate nested properties within the configuration using dot-notation. This includes `parseConfigPath` for parsing and validating paths to prevent vulnerabilities like prototype pollution by blocking dangerous keys (e.g., `__proto__`), as well as functions for `setConfigValueAtPath`, `unsetConfigValueAtPath`, and `getConfigValueAtPath` for managing configuration values at specific paths. These utilities ensure that configurations are well-structured and securely managed, with tests located in `src/config/config-paths.test.ts` and the implementation in `src/config/config-paths.ts`. A critical validation also prevents multiple agents from sharing the same `agentDir`, which could lead to state collisions, a scenario tested in `src/config/config.multi-agent-agentdir-validation.test.ts`.

Cron Job Management System



The OpenClaw cron job management system provides facilities for defining, scheduling, and executing automated tasks within the OpenClaw environment. It supports various scheduling paradigms, manages the lifecycle of cron jobs, ensures their persistence, and logs their execution.

Job definition and normalization are handled by functions such as `normalizeCronJobCreate` in `src/cron/normalize.ts`. These functions process raw input, coercing types, applying default values, and handling legacy formats to ensure that all cron job configurations adhere to a consistent structure. This includes sanitizing agent IDs, converting schedule fields like `atMs` into standardized `at` ISO strings, and consolidating delivery-related fields that might have originated from older payload structures. The `migrateLegacyCronPayload` function in `src/cron/payload-migration.ts` specifically addresses backward compatibility for channel and provider fields within job payloads.

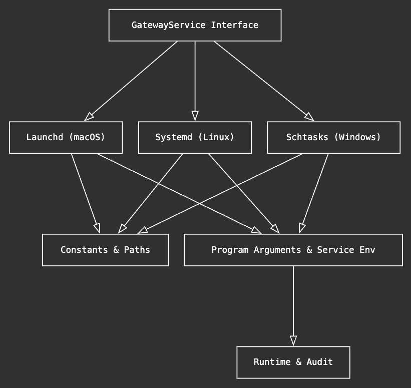
For scheduling, the system uses `computeNextRunAtMs` in `src/cron/schedule.ts` to determine the next execution time for a job. This function supports "at" schedules for one-time execution, "every" schedules for recurring intervals, and standard cron expressions for complex periodic tasks. Timezone considerations are integrated into the scheduling logic to ensure accurate execution times.

Execution of cron jobs can involve "isolated agent turns," where an agent is configured and run as a separate, self-contained process. The `runCronIsolatedAgentTurn` function, exported by `src/cron/isolated-agent.ts`, manages this process. It handles agent and model configuration, workspace and session management, prompt preparation (including security measures against prompt injection for external content), and the eventual execution of the agent. After execution, it processes the agent's output, extracting summaries and handling message delivery. The `resolveCronDeliveryPlan` function in `src/cron/delivery.ts` determines how an agent's response should be delivered, specifying the mode, channel, and recipient.

The `CronService` in `src/cron/service` manages the overall lifecycle of cron jobs. It offers capabilities to add, update, remove, start, and stop jobs. The service handles persistence, loading job definitions from a store at startup and saving changes to disk to ensure durability. Concurrency control mechanisms are in place to prevent race conditions when multiple operations attempt to modify the job store. The `CronService` also emits events to allow external components to react to changes in job status and execution.

Run logging is a crucial part of the system, provided by utilities in `src/cron/run-log.ts`. These utilities manage the storage and retrieval of cron job execution logs, storing them in a line-delimited JSON format. Logs are automatically pruned to manage file size and retain a specified number of recent entries, ensuring that historical data is available without consuming excessive storage.

Daemon Service Management



The daemon service management system handles the installation, uninstallation, starting, stopping, and status querying of background services, specifically for the gateway and node components of OpenClaw. It provides a platform-agnostic interface that abstracts the underlying operating system-specific mechanisms, such as `launchd` on macOS, `systemd` on Linux, and Scheduled Tasks on Windows.

The core of the daemon management is defined by the `GatewayService` interface in `src/daemon/service.ts`, which offers a unified API for service lifecycle operations. Implementations of this interface dynamically adapt to the operating system:

- On macOS, the `launchd` agents are managed through `src/daemon/launchd.ts`. This involves generating and interacting with `.plist` files and using the `launchctl` command-line utility. The system handles legacy agent cleanup and robust parsing of `launchctl` output.

- For Linux environments, `systemd` user services are managed by `src/daemon/systemd.ts`. This module handles the installation, uninstallation, and control of `.service` unit files, interacting with `systemctl`. It also includes diagnostics for `systemd` availability and management of user linger settings.

- On Windows, Scheduled Tasks are utilized via `src/daemon/schtasks.ts`. This involves generating and parsing batch scripts (`.cmd`) that execute the service logic, and interacting with the `schtasks` command. This module also provides utilities for proper command-line argument parsing and quoting on Windows.

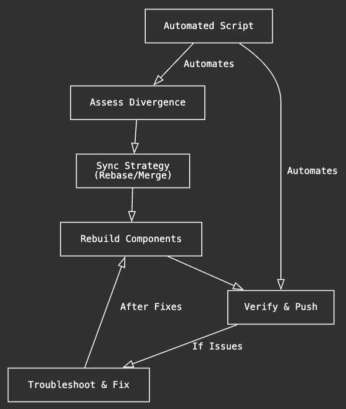
Central to these operations is consistent naming and configuration. The system uses `src/daemon/constants.ts` to define default service names and generate profile-specific identifiers, allowing for multiple instances of services with distinct configurations. Path resolution, handled by `src/daemon/paths.ts`, ensures that service files and state directories are located correctly across platforms, supporting environment variable overrides and tilde expansion for user home directories.

The determination of program arguments and the working directory for daemon services is critical for their correct execution. The `src/daemon/program-args.ts` module resolves command-line arguments for both gateway and node services, intelligently handling different JavaScript runtimes (Node.js, Bun) and execution modes (development vs. production). It manages complexities like `npx` shims and `pnpm` symlinks to ensure stable and correct entry point resolution.

Environment variables are constructed for services through `src/daemon/service-env.ts`, dynamically building the `PATH` environment variable and populating OpenClaw-specific variables necessary for service operation. The `src/daemon/node-service.ts` module specifically configures `GatewayService` for node services by injecting node-specific environment variables into all service operations.

To ensure service health and aid in troubleshooting, the daemon management system includes diagnostic tools. `src/daemon/diagnostics.ts` provides utilities for reading and identifying gateway-related errors from log files, while `src/daemon/inspect.ts` detects and lists "extra" or legacy gateway services that might be running, offering cleanup hints. Furthermore, `src/daemon/service-audit.ts` audits gateway service configurations for potential issues related to runtime, environment variables, and platform-specific settings, identifying misconfigurations and providing recommendations for optimal setup. The resolution and compatibility checking of Node.js runtime paths are handled by `src/daemon/runtime-paths.ts`, ensuring the daemon operates with a supported Node.js version.

Clawdbot Development Environment Management



The `Clawdbot` development environment facilitates the maintenance and updating of local developer setups. The primary workflow, detailed in `.agent/workflows/update_clawdbot.md`, focuses on keeping a local fork synchronized with the upstream repository. This involves assessing divergence between branches using Git commands and applying changes through either `rebase` or `merge` strategies. The `rebase` approach is generally preferred for its ability to maintain a linear project history.

After synchronizing, the environment requires rebuilding various components. This includes reinstalling dependencies, building TypeScript files, and compiling UI assets. A dedicated script or command automates the process of rebuilding the macOS application, installing it, and launching it for verification. To ensure the updated environment functions correctly, developers can run health checks and agent-specific commands.

A significant aspect of the `Clawdbot` development environment involves troubleshooting macOS and Swift compatibility issues. The workflow includes specific guidance for identifying and resolving problems related to Swift 6.2 and macOS 26 SDK, often involving codebase searches and targeted text replacements. An automated shell script is also available to streamline the entire synchronization, build, and verification process, reducing manual steps and potential errors. Further details on general agent functionality can be found in Core Agent Development and Workflow.

Coding Agent Extensions and Prompts

Path	Type	Description	Purpose/Workflow
`/openclaw/openclaw/.pi/extensions/diff.ts`	Extension	`/diff` command: shows modified/deleted/new files from `git status` and opens selected file in VS Code's diff view.	Git interaction (viewing changes)
`/openclaw/openclaw/.pi/extensions/files.ts`	Extension	`/files` command: lists all files the model has read/written/edited in the active session branch. Selecting a file opens it in VS Code.	File tracking and quick access
`/openclaw/openclaw/.pi/extensions/prompt-url-widget.ts`	Extension	Displays a widget in the UI for GitHub PR/Issue URLs found in the prompt, showing title and author, and sets the session name.	UI enhancement, contextual information, session management
`/openclaw/openclaw/.pi/extensions/redraws.ts`	Extension	`/tui` command: shows TUI redraw statistics.	UI debugging/performance monitoring
`/openclaw/openclaw/.pi/prompts/cl.md`	Prompt	Guides auditing changelog entries for all commits since the last release.	Changelog management and release preparation
`/openclaw/openclaw/.pi/prompts/is.md`	Prompt	Guides analysis of GitHub issues (bugs or feature requests), including tracing code paths and proposing solutions.	Issue analysis and solution planning
`/openclaw/openclaw/.pi/prompts/landpr.md`	Prompt	Guides the end-to-end workflow for landing a GitHub Pull Request (merging with proper Git workflow).	PR management (merging)
`/openclaw/openclaw/.pi/prompts/reviewpr.md`	Prompt	Guides a thorough review of a GitHub Pull Request, producing a recommendation without merging.	PR management (code review)

The `Clawdbot` development environment extends the capabilities of a coding agent through various integrations and structured prompts. These extensions facilitate interaction with Git, track file interactions, process GitHub URLs, and provide terminal user interface (TUI) statistics. The structured prompts guide the agent through software development workflows such as changelog auditing, GitHub issue analysis, and comprehensive management of GitHub pull requests.

The `.pi/extensions` directory houses several TypeScript modules that augment the coding agent's functionality:

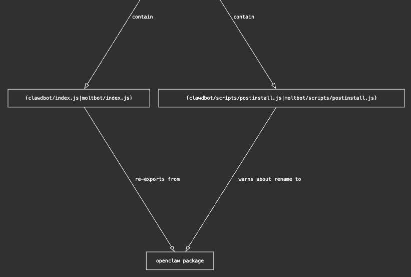
- The `diff` command, defined in `.pi/extensions/diff.ts`, allows users to interactively view changes detected by `git status`. It can open selected files in VS Code's diff view or directly open untracked files.
- The `files` command, implemented in `.pi/extensions/files.ts`, tracks files that have been read, written, or edited by the agent during a session. This tracking is based on analyzing `toolCall` and `toolResult` messages, presenting the aggregated file activity in an interactive TUI, and enabling users to open these files in VS Code.
- The prompt URL widget, described in `.pi/extensions/prompt-url-widget.ts`, enhances the agent's context by recognizing GitHub PR or issue URLs in user prompts. It fetches relevant metadata (title, author) using the `gh` CLI tool, displays this information in a UI widget, and updates the session name accordingly. This feature also ensures consistency across sessions by rebuilding the widget and session name upon session changes.
- TUI redraw statistics, managed by `.pi/extensions/redraws.ts`, provide a `tui` command that calculates and displays the number of full redraws performed by the Text User Interface, offering insights into TUI performance.

The `.pi/prompts` directory contains markdown files that serve as structured guides for various software development tasks:

- `.pi/prompts/cl.md` outlines a process for manually auditing changelog entries before a software release. This involves identifying commits since the last release, reviewing changelog entries across specific packages, checking for skip conditions, enforcing cross-package duplication rules, and generating a report of missing or duplicated entries.
- `.pi/prompts/is.md` provides a procedure for analyzing GitHub issues. For bug analysis, it focuses on tracing root causes and proposing fixes, while for feature requests, it suggests implementation approaches and lists affected files. The prompt explicitly restricts the agent to analysis and proposal, preventing direct implementation unless specifically requested.
- `.pi/prompts/landpr.md` details a workflow for merging a GitHub Pull Request into the main branch. This comprehensive guide covers identifying PR metadata, synchronizing the main branch, locally checking out and rebasing the PR, fixing conflicts, implementing fixes, adding tests, updating the changelog, running pre-merge checks, committing changes, pushing the updated branch, merging the PR on GitHub, and cleaning up temporary branches.
- `.pi/prompts/reviewpr.md` describes a systematic process for reviewing GitHub Pull Requests without performing any merging or code modifications. It covers identifying PR meta-information, reading the description and diff, validating the change's necessity and implementation quality, evaluating tests, and suggesting follow-up refactors. The output of this process is a structured review with a clear recommendation.

Legacy Package Compatibility

{clawdbot|moltbot} packages

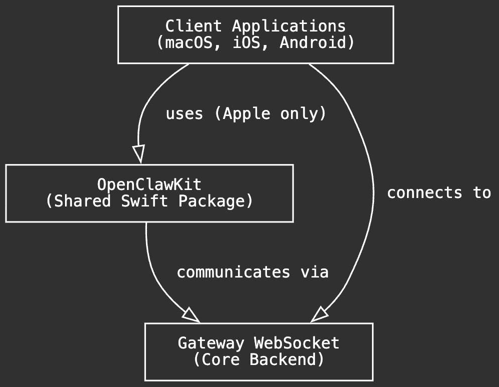

contain
contain
{clawdbot/index.js|moltbot/index.js}
{clawdbot/scripts/postinstall.js|moltbot/scripts/postinstall.js}
re-exports from
warns about rename to
openclaw package

This section addresses the management of legacy package names used in the OpenClaw project to ensure a smooth transition to the current `openclaw` package name. It details how the project maintains compatibility for older package references, specifically `clawdbot` and `moltbot`, while guiding users towards the updated naming convention.

The `packages` directory is central to this effort, serving to re-export the `openclaw` module's public APIs through the deprecated package names. For instance, the `packages/clawdbot/index.js` and `packages/moltbot/index.js` files effectively mirror the functionality of `openclaw`, ensuring that any existing code relying on these older package names continues to function without modification.

To facilitate the migration, post-installation scripts are included within these legacy package directories. Specifically, `packages/clawdbot/scripts/postinstall.js` and `packages/moltbot/scripts/postinstall.js` issue warning messages to users via `console.warn` upon installation. These warnings inform developers about the package renaming and encourage them to update their dependencies to `openclaw`. This approach ensures backwards compatibility for existing deployments while actively promoting the adoption of the new, standardized package name.

Client Applications



OpenClaw's client applications span Android, iOS, and macOS platforms, providing users with functionalities such as Canvas, Chat, and Camera. These applications leverage a shared Swift package for core logic and UI components, promoting code reuse and consistency across Apple's ecosystems. All client applications connect to a `Gateway WebSocket` to facilitate communication and expose device capabilities.

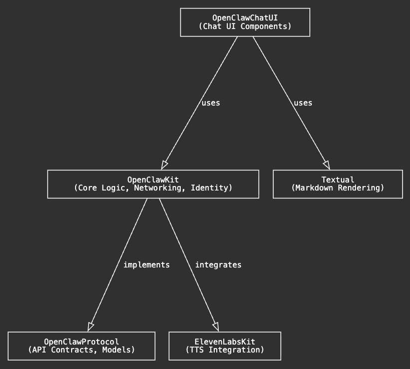
The macOS application, found in `apps/macos`, operates as a menu bar application, offering a command-line interface (CLI) and utilizing an Inter-Process Communication (IPC) library. It manages its application state, handles software updates via `Sparkle`, and integrates OAuth for authentication. This application also provides robust camera and audio capture services, and includes mechanisms for discovering OpenClaw gateways on both local and wide-area networks.

The iOS application, located in `apps/ios`, provides a development scaffold for internal use. It manages application state and communicates with the OpenClaw gateway. This includes handling deep links to specific application features, and offering various device functionalities like camera, screen recording, location services, and voice interaction. The development workflow for the iOS app is supported by `fastlane` for automation.

The Android application, situated in `apps/android`, is a modern node app built using Kotlin and Jetpack Compose. It maintains a persistent connection to the `Gateway WebSocket` via a foreground service, ensuring continuous operation. This application exposes Canvas, Chat, and Camera functionalities, integrating with the OpenClaw ecosystem to provide device capabilities on the Android platform.

A foundational element for the Apple client applications is the shared Swift package, `OpenClawKit`, located in `apps/shared`. This package centralizes core functionalities and UI components. It defines common communication protocols, networking logic, device identity management, and structured command sets that are utilized by both the iOS and macOS applications. `OpenClawKit` also includes components for audio processing and flexible data types, ensuring a cohesive experience across platforms.

Cross-Platform Shared Swift Package (OpenClawKit)



The `OpenClawKit` Swift package, located primarily in `apps/shared/OpenClawKit`, provides a shared foundation for OpenClaw's iOS and macOS client applications. It encapsulates core functionalities and UI components, enabling consistent behavior and appearance across different Apple platforms.

The package defines three main logical products: `OpenClawProtocol`, `OpenClawKit`, and `OpenClawChatUI`. `OpenClawProtocol` establishes the fundamental definitions and protocols for client-server communication. `OpenClawKit` contains core business logic, including networking, device identity management, and structured command definitions. `OpenClawChatUI` provides the complete set of user interface components and logic specifically for chat functionalities.

For networking, the kit manages WebSocket connections to a gateway, handles authentication, and ensures automatic reconnection. It also implements TLS certificate pinning for secure communication and defines structures for a bridge communication protocol for inter-process communication. Server-initiated push messages from the gateway are processed to update application states.

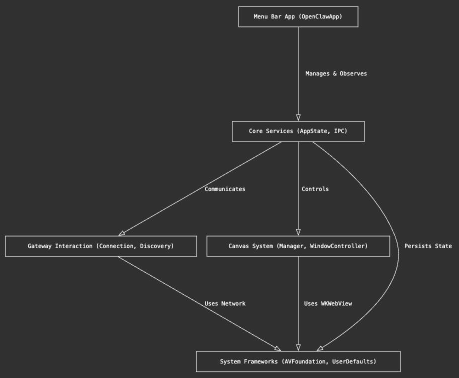
Device identity management includes generating and storing cryptographic key pairs, handling authentication tokens for different device roles, and providing unique identifiers for application instances. Service discovery mechanisms like Bonjour are supported for locating gateways.

The command structures within `OpenClawKit` define various application capabilities and actions. These include commands for camera operations (like `snap` and `clip`), canvas controls (such as `present`, `hide`, `navigate`, and `snapshot`), and system-level commands for running processes or sending notifications. Specialized commands for interacting with the A2UI framework are also defined, enabling agents to dynamically generate and update user interfaces.

Audio processing capabilities include protocols for streaming audio playback and tools for constructing prompts in "Talk Mode," allowing for voice-controlled interactions. The kit also integrates with external services for text-to-speech functionalities.

Finally, `OpenClawKit` employs flexible data types, notably `AnyCodable` in `apps/shared/OpenClawKit/Sources/OpenClawProtocol/AnyCodable.swift`, to handle heterogeneous JSON payloads. This design choice accommodates the dynamic nature of messages within the OpenClaw ecosystem, ensuring that varying data structures can be encoded and decoded robustly. Helper functions are also available for processing "wizard" data, which often involves dynamic steps and options during user onboarding or configuration.

macOS Application Architecture and Features


Menu Bar App (OpenClawApp)
Manages & Observes
Core Services (AppState, IPC)
Communicates
Controls
Persists State
Gateway Interaction (Connection, Discovery)
Canvas System (Manager, WindowController)
Uses Network
Uses WKWebView
System Frameworks (AVFoundation, UserDefaults)

The OpenClaw macOS application provides a comprehensive interface for interacting with the OpenClaw ecosystem, integrating closely with the macOS environment. Its architecture supports a menu bar presence, a command-line interface (CLI), inter-process communication (IPC) for system-level interactions, robust state management, and multi-platform gateway discovery.

The core application logic, found within `apps/macos/Sources/OpenClaw`, manages global application settings and user preferences through an observable `AppState` object. This state controls various aspects, including application behavior (e.g., `isPaused`, `launchAtLogin`), voice wake functionality, UI appearance (e.g., `iconAnimationsEnabled`, `showDockIcon`), and connection parameters. Settings are persisted using `UserDefaults` and can be dynamically updated via a `ConfigFileWatcher` that monitors external configuration changes. Authentication for services, such as Anthropic, is handled via a dedicated SwiftUI view, `AnthropicAuthControls`, which orchestrates the OAuth 2.0 PKCE flow, displaying authentication status and securely managing credentials. Agent-generated events are logged and displayed through an `AgentEventStore` and `AgentEventsWindow`, providing a real-time view of agent activity.

The application includes an `openclaw` CLI, with installation prompted to users by `CLIInstallPrompter`. The `CLIInstaller` handles the execution of installation scripts and determines the installed location. Media capture services, provided by a `CameraCaptureService` actor, allow the application to list devices, snap photos, and record videos, integrating with `AVFoundation` for access and transcoding.

A key feature is the Canvas system, which enables the dynamic display of web-based content within the application. The `CanvasManager` controls the lifecycle of `CanvasWindowController` instances, managing the display, hiding, and evaluation of JavaScript within the canvas, and providing snapshot capabilities. It also observes `GatewayConnection` events to automatically navigate to A2UI content. A custom URL scheme, `"openclaw-canvas"`, allows the application to serve internal web resources, with a `CanvasSchemeHandler` preventing directory traversal and serving scaffold or welcome pages for sessions. The `CanvasWindowController` also handles navigation, managing deep links and redirecting external URLs.

Inter-process communication (IPC) is a critical component, defined by structures in `apps/macos/Sources/OpenClawIPC`. This system allows various parts of the application to communicate requests and responses, covering notifications, permission management, shell command execution, canvas control, node interactions, and camera/screen recording actions. Requests are defined as an enum, `Request`, enabling a type-safe and extensible API, while responses follow a standardized `Response` format.

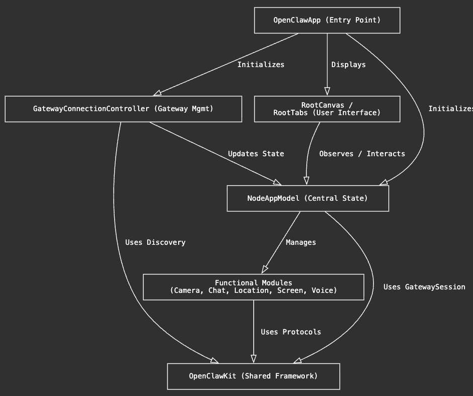
Gateway discovery is managed by the code within `apps/macos/Sources/OpenClawDiscovery`. The `GatewayDiscoveryModel` identifies available OpenClaw gateways using both Bonjour for local network discovery and wide-area discovery techniques, leveraging DNS queries via tools like `dig` and `tailscale status`. It processes TXT records for detailed gateway information and identifies the local device to filter discovery results. This model continuously aggregates and sorts discovered gateways, making them available for user interaction.

The `openclaw-mac` command-line utility, located in `apps/macos/Sources/OpenClawMacCLI`, provides functionalities for connecting to, discovering, and configuring OpenClaw Gateways. It parses command-line arguments and dispatches them to dedicated subcommands. The `connect` subcommand handles establishing and maintaining connections to gateways, performing health checks, and storing connection snapshots. The `discover` subcommand utilizes the `GatewayDiscoveryModel` to find available gateways and formats the output. An interactive `wizard` subcommand guides users through gateway configuration, communicating with the gateway via a `GatewayWizardClient` to manage device identity and security.

The application also integrates with `Sparkle` for update management, which is configured in the `Package.swift` file (`apps/macos/Package.swift`). This manifest defines the overall structure of the macOS application, including its dependencies on external libraries for menu bar functionality, subprocess execution, logging, and UI automation, as well as internal shared components like `OpenClawKit` and `Swabble`. The development, signing, and packaging process for the macOS application is detailed in `apps/macos/README.md`, outlining the use of scripts for building, code signing, and distributing the application. This includes procedures for managing signing identities, performing Team ID audits for consistency across embedded binaries, and handling development-only workarounds for library validation issues.

The gateway protocol itself is defined in `apps/macos/Sources/OpenClawProtocol`, providing Swift `Codable` and `Sendable` data structures for error codes, connection parameters, server responses, and various API-specific models. These models ensure structured communication for agents, sessions, cron jobs, logging, and device pairing.

iOS Application Development and Capabilities


OpenClawApp (Entry Point)
Initializes
Displays
Initializes
GatewayConnectionController (Gateway Mgmt)
RootCanvas / RootTabs (User Interface)
Updates State
Observes / Interacts
NodeAppModel (Central State)
Uses Discovery
Manages
Functional Modules (Camera, Chat, Location, Screen, Voice)
Uses GatewaySession
Uses Protocols
OpenClawKit (Shared Framework)

The OpenClaw iOS application provides a scaffold for internal development, integrating various functionalities for user interaction and gateway communication. The setup documentation, located in `apps/ios/README.md`, details prerequisites for linting and formatting, Xcode project generation using `xcodegen`, and the utilization of `fastlane` for automation. This `fastlane` integration is further explained in `apps/ios/fastlane/SETUP.md`, covering installation, API key generation for App Store Connect, and environment variable configuration to facilitate tasks such as beta releases.

Application state and lifecycle management are centralized within the `NodeAppModel` class, found in `apps/ios/Sources/Model/NodeAppModel.swift`. This class orchestrates connections to the `OpenClawKit` gateway, dispatches commands received from the gateway to specific handlers, and integrates various device services. The application's entry point, `OpenClawApp` in `apps/ios/Sources/OpenClawApp.swift`, sets up this model, along with a `GatewayConnectionController`, and handles deep links and changes in the application's scene phase.

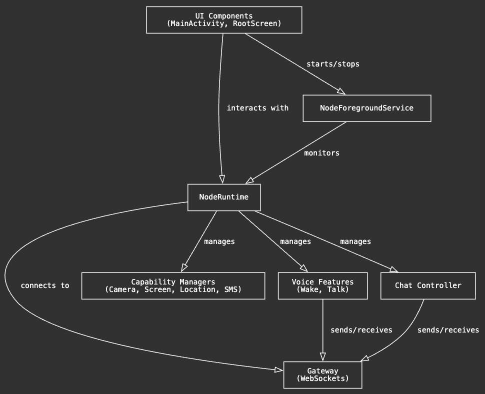
The user interface (UI) is structured around `RootCanvas` in `apps/ios/Sources/RootCanvas.swift`, which serves as the main display area, and `RootTabs` in `apps/ios/Sources/RootTabs.swift`, which manages tab-based navigation for "Screen," "Voice," and "Settings." These UI components interact with the `NodeAppModel` and other managers to reflect the application's status and user preferences. Visual feedback, such as a `StatusPill` for connection states and a `VoiceWakeToast` for voice commands, is provided through components defined in `apps/ios/Sources/Status`.

Key functionalities are modularized into dedicated directories:

- **Camera**: The `apps/ios/Sources/Camera` directory handles photo and video capture, device listing, and media format conversion.
- **Chat**: The `apps/ios/Sources/Chat` directory provides a chat interface, managing message sending, history, and real-time event streaming through a `GatewayNodeSession`.
- **Gateway**: The `apps/ios/Sources/Gateway` directory manages gateway discovery, connection, and secure persistent storage of settings and credentials, including the implementation of the TLS Handshake and auto-connection mechanisms.
- **Location**: The `apps/ios/Sources/Location` directory handles iOS location services, including authorization and current location retrieval with caching.
- **Screen**: The `apps/ios/Sources/Screen` directory manages web content display via `WKWebView`, supporting deep links, JavaScript execution, and screen recording.
- **Settings**: The `apps/ios/Sources/Settings` directory manages application settings such as network configuration, gateway connectivity, voice wake features, and camera access, persisting these preferences.
- **Voice**: The `apps/ios/Sources/Voice` directory implements voice interaction features, including continuous wake word detection, speech-to-text, text-to-speech, and associated UI components. Session key management is handled by `SessionKey` in `apps/ios/Sources/SessionKey.swift` for consistent session identification.

A comprehensive testing strategy, located in `apps/ios/Tests`, covers critical components, including `NodeAppModel` for scene phase handling (`apps/ios/Tests/AppCoverageTests.swift`), `VoiceWakeManager` for simulator behavior, `CameraController` for input clamping (`apps/ios/Tests/CameraControllerClampTests.swift`) and error descriptions (`apps/ios/Tests/CameraControllerErrorTests.swift`), `DeepLinkParser` for URL parsing (`apps/ios/Tests/DeepLinkParserTests.swift`), and various gateway and screen-related functionalities.

Android Application Core Functionalities



The OpenClaw Android node application provides core functionalities for connecting to the OpenClaw ecosystem, exposing device capabilities, and managing user interactions. This application, built with Kotlin and Jetpack Compose, establishes a persistent connection to a `Gateway WebSocket` and handles various aspects of the user experience.

The application's entry point, handled by `MainActivity`, sets up the user interface and manages permissions necessary for its operation. A `NodeForegroundService` maintains a persistent background connection to the gateway, ensuring continuous operation even when the app is not actively in use. This service uses a persistent notification to inform the user of its active status.

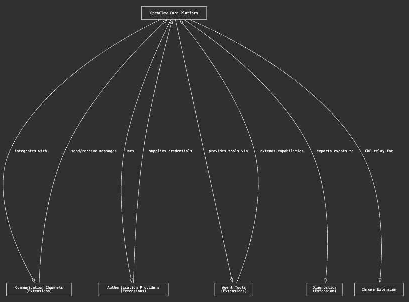
Central to the application's functionality is the `NodeRuntime`, which orchestrates communication with the gateway and manages various device capabilities. It handles `GatewayDiscovery`, connecting to and refreshing gateway connections, and configuring secure TLS communication. This runtime integrates managers for different device capabilities:

- **Camera Capabilities:** The `CameraCaptureManager` facilitates image capture (`snap`) and video recording (`clip`), including necessary permission checks and image processing functions.
- **Canvas Integration:** The `CanvasController` manages a `WebView` component, allowing the application to render UI elements and interactive experiences on a web canvas. This includes navigation, evaluation of JavaScript, and capturing snapshots of the canvas content.
- **Location Services:** The `LocationCaptureManager` provides access to device location data, incorporating permission management and different location modes.
- **Screen Recording:** The `ScreenRecordManager` manages the initiation and control of screen recording functionality.
- **SMS Management:** The `SmsManager` handles sending SMS messages, including required permission checks.

The application also includes robust chat and voice interaction features. The `ChatController` manages chat sessions, allowing users to load, send, and switch between conversations. A shared `main` session key ensures cross-platform compatibility for chat functionality. Voice features are managed by `VoiceWakeManager` for wake word detection and `TalkModeManager` for "Talk Mode," which orchestrates speech-to-text (STT) and text-to-speech (TTS) processes. The `TalkDirectiveParser` interprets embedded JSON directives within spoken commands, enabling rich voice interactions.

User preferences and sensitive data, such as instance IDs and display names, are securely persisted using `SecurePrefs`. The application's UI is built using Jetpack Compose, providing a modern and responsive user experience. Key UI components include `RootScreen`, which integrates the Canvas view with overlays and sheets, a `StatusPill` to display connection and voice activity, and `TalkOrbOverlay` for visual feedback on AI assistant interactions. Dedicated `ChatSheet` and `SettingsSheet` components provide interfaces for managing chat conversations and application settings, respectively. The `apps/android/app/src/main/java/ai/openclaw/android/ui/chat` directory contains further details on chat-specific UI components.

OpenClaw Extensions and Integrations



OpenClaw's capabilities are expanded through various extensions and integrations, including communication channel plugins, authentication providers, and agent tools. The system also supports a Chrome extension for the OpenClaw Gateway, which facilitates browser automation by utilizing the Chrome DevTools Protocol (CDP).

Communication channel plugins enable OpenClaw to integrate with diverse messaging platforms, managing aspects such as account configuration, message handling, security policies, onboarding, and runtime monitoring. Examples include plugins for BlueBubbles, Discord, Feishu, Google Chat, iMessage, LINE, and Matrix. These plugins allow OpenClaw agents to send and receive messages, manage chat participants, and handle reactions within these external communication environments. Each plugin typically defines its own configuration schemas, message processing logic, and methods for monitoring the connection to the respective platform. For instance, the BlueBubbles plugin, located in `extensions/bluebubbles`, manages message actions like reactions, edits, and unsend functionality, along with attachment handling and chat management. Similarly, the Discord plugin in `extensions/discord` supports direct, channel, and thread chats, polls, reactions, and media. The Feishu plugin in `extensions/feishu` offers comprehensive integration, including document, drive, wiki, and permission management capabilities, with extensive tools for messaging and real-time event processing. Google Chat integration, found in `extensions/googlechat`, handles account configurations, API interactions, and webhook processing for seamless communication. The iMessage plugin in `extensions/imessage` provides account management, message handling, and security policies for iMessage interactions. The LINE extension in `extensions/line` enables interaction with the LINE Messaging API, supporting rich card message generation and various message types. The Matrix plugin in `extensions/matrix` provides onboarding, pairing, and capabilities for Matrix accounts. The Mattermost plugin in `extensions/mattermost` integrates with Mattermost for communication, including message sending and status monitoring. Microsoft Teams integration in `extensions/msteams` supports adaptive cards, file sharing, and various messaging capabilities. Nextcloud Talk integration in `extensions/nextcloud-talk` manages account configurations, webhook processing, and outbound messaging.

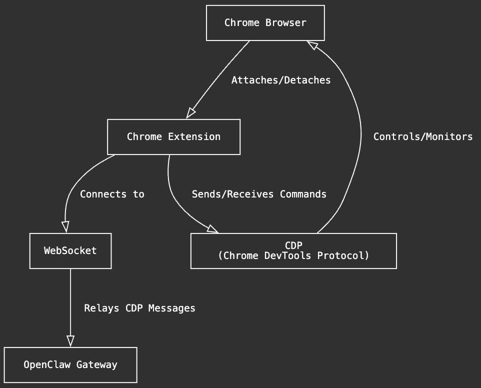
Authentication provider plugins facilitate secure integration with external services by handling OAuth authentication flows. These plugins manage token exchange, credential discovery, and often implement Proof Key for Code Exchange (PKCE) for enhanced security. For example, the Google Antigravity (Cloud Code Assist) OAuth provider plugin, located in `extensions/google-antigravity-auth`, orchestrates the entire OAuth 2.0 process, from PKCE code generation to token exchange and user/project information retrieval. The Google Gemini CLI (Google Code Assist) OAuth provider plugin in `extensions/google-gemini-cli-auth` handles authentication, automatically extracting credentials or allowing environment variable overrides, and supports both local and manual authorization flows. The MiniMax Portal authentication plugin in `extensions/minimax-portal-auth` uses a device code flow with PKCE to authenticate with MiniMax, subsequently configuring relevant models.

Agent tool plugins extend OpenClaw's capabilities for automated tasks, allowing agents to perform specialized operations. The `llm-task` tool, found in `extensions/llm-task`, enables agents to execute JSON-only Large Language Model (LLM) tasks, such as drafting or summarizing, with optional JSON Schema validation for output. This tool sends prompts and input to an LLM, processes the JSON output, and validates it if a schema is provided. The `lobster` tool, located in `extensions/lobster`, allows OpenClaw to execute and resume structured workflows, including those requiring human approval, thereby leveraging `lobster`'s deterministic and resumable nature. This integration permits `lobster` pipelines to invoke other OpenClaw tools. The `memory-core` plugin in `extensions/memory-core` provides foundational memory functionalities, while the `memory-lancedb` plugin in `extensions/memory-lancedb` implements a LanceDB-based memory extension with vector search for managing long-term agent memory.

For diagnostics, the OpenTelemetry (OTel) plugin in `extensions/diagnostics-otel` allows OpenClaw to export diagnostic events as traces, metrics, and logs to an OTel collector. This plugin configures OTLP exporters for HTTP/protobuf and captures events such as model usage, webhook processing, and session status, providing visibility into the system's operation.

The Chrome extension, located in `assets/chrome-extension`, facilitates browser automation by establishing a local Chrome DevTools Protocol (CDP) relay server. This enables OpenClaw Gateway to attach to and detach from Chrome tabs, forward CDP commands and events, and configure relay port settings. The `background.js` script in `assets/chrome-extension/background.js` manages the WebSocket connection to the relay server and handles the forwarding of CDP commands and events. The `options.html` and `options.js` files, in `assets/chrome-extension/options.html` and `assets/chrome-extension/options.js` respectively, provide a user interface for configuring the relay port and checking the relay server's reachability.

Chrome Extension for OpenClaw Gateway



The Chrome extension facilitates the interaction between the `OpenClaw Gateway` and the Chrome browser, enabling automation and inspection of Chrome tabs through the Chrome DevTools Protocol (CDP). It functions as a local CDP relay server, allowing the `OpenClaw Gateway` to remotely control browser actions.

The extension's primary components include a background script, an options page, and a `README.md` file. The background script, located at `assets/chrome-extension/background.js`, manages the WebSocket connection to an external relay server. It handles the attachment and detachment of the CDP debugger to browser tabs, forwarding CDP commands from the relay server to the browser and sending CDP events from the browser back to the relay. This script also provides visual feedback through the extension's badge and title, indicating connection status and debugger state. The `DEFAULT_PORT` constant within this script defines the default communication port for the relay server.

Users can configure the relay server's port through the extension's options page, defined in `assets/chrome-extension/options.html` and managed by `assets/chrome-extension/options.js`. This page allows for port number validation, display of the relay URL, and status messages, including checks for the relay's reachability. The `clampPort` function ensures that entered port numbers are within a valid range, and the `checkRelayReachable` function performs an asynchronous check of the relay server's connectivity. User preferences for the `relayPort` are persisted using `chrome.storage.local`.

Installation and usage instructions, including building the `OpenClaw Gateway` and installing the extension in developer mode, are provided in `assets/chrome-extension/README.md`. Once installed, users can attach or detach OpenClaw to a browser tab by clicking the extension icon.

Communication Channel Plugins

Feature	BlueBubbles	Discord	Feishu	Google Chat	iMessage	L
Account Management	Multi-account, dynamic config	Multi-account, token/env-based config	Multi-account, dynamic config	Multi-account, service account/token-based config	Multi-account, dynamic config	M b
Messaging Capabilities	Direct, group chats, media, reactions, edit, unsend, reply, effects, group management	Direct, channel, thread chats, polls, reactions, threads, media, native commands	Direct, group chats, media, reactions, native commands, block streaming	Direct, group, thread chats, reactions, threads, media, block streaming	Direct, group chats, media	D bl
Security - DM Policy	Pairing, allowlist (normalized handles)	Pairing, allowlist (user IDs, mentions)	Configurable (open, pairing, allowlist)	Pairing, allowlist (user/space IDs)	Pairing, allowlist (handles)	P
Security - Group Policy	Configurable (open, allowlist, disabled)	Configurable (open, allowlist, disabled)	Configurable (open, allowlist, disabled)	Configurable (open, allowlist, disabled)	Configurable (open, allowlist, disabled)	C al

Onboarding & Setup	Guided setup for server URL, password, webhook	Guided setup for bot token (env or direct)	Guided setup for app ID, app secret, connection mode	Guided setup for service account (file/json/env)	Guided setup for CLI/DB paths	G ac or
Message Handling	Rich target resolution, text chunking	Rich target resolution, message actions, streaming, text chunking, markdown support	Rich target resolution, message actions, markdown processing, flex messages, rich messages	Rich target resolution, message actions, streaming, markdown chunking	Rich target resolution, text chunking	Ri m m m
Runtime Monitoring	Status, last error, probe endpoint	Status, bot info, audit permissions, probe endpoint	Status, last error, probe endpoint	Status, audience info, webhook paths, probe endpoint	Status, CLI/DB paths, last error	St er
Special Features	Webhook requests	Polls, threads, native commands	Bitable, Doc, Drive, Perm, Wiki tools, rich messages	Threads	-	Ri (c ca

OpenClaw integrates with various communication platforms through a plugin architecture, enabling agents to send and receive messages, manage accounts, enforce security policies, and monitor real-time events. Each communication channel is implemented as a plugin, providing a standardized interface for OpenClaw to interact with external services. These plugins handle the specifics of each platform's API, authentication mechanisms, and message formats, abstracting these complexities from the core OpenClaw system.

Account management is a core function of these plugins, encompassing tasks such as listing configured accounts, resolving default accounts, and managing account credentials. For instance, the BlueBubbles plugin within `extensions/bluebubbles` facilitates the configuration and management of BlueBubbles accounts, including merging configurations and resolving account details. Similarly, the Discord plugin, found in `extensions/discord`, handles Discord account settings, while the Feishu/Lark integration in `extensions/feishu` supports multi-account setups and validates configurations using `zod` schemas. Google Chat, implemented in `extensions/googlechat`, centralizes authentication and account configurations. iMessage, covered in `extensions/imessage`, and LINE, in `extensions/line`, also provide robust account management features including credential handling and configuration persistence. The Matrix plugin, in `extensions/matrix`, manages Matrix account lifecycles and credential storage, and the Mattermost plugin, in `extensions/mattermost`, provides mechanisms for account resolution and configuration.

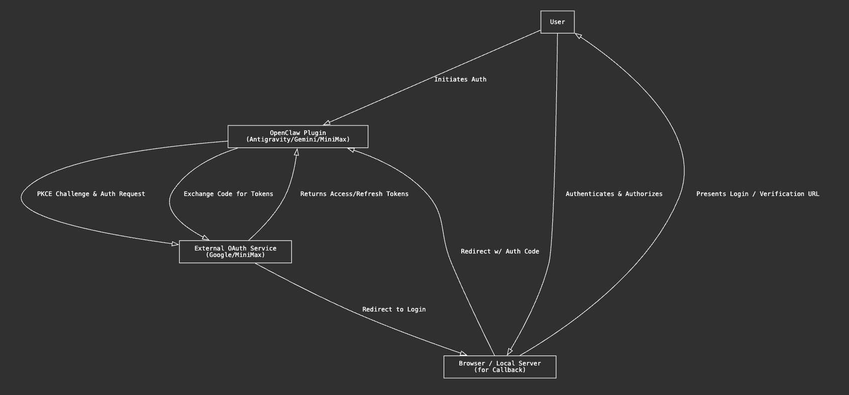
Message handling is another critical aspect, with plugins defining how text, media, and interactive elements are sent and received. For example, the BlueBubbles plugin provides functions for sending text messages, attachments, and reactions, along with managing chat states like typing indicators. The Discord plugin supports sending text, media, and polls, utilizing its runtime to interact with the Discord API. Feishu handles diverse message types, including rich cards, and provides utilities for media uploads and mention parsing. Google Chat facilitates sending and updating messages, uploading attachments, and managing reactions. LINE offers extensive capabilities for sending various message types, including `flexMessage` and `templateMessage`, and supports rich card generation through specific commands. The Matrix plugin manages message operations including text, media, polls, reactions, and thread management. Mattermost handles sending messages and media, including file uploads, and processes inbound messages via WebSocket connections.

Security policies and access control are integral to each plugin, ensuring that agents interact with channels according to defined rules. Plugins implement mechanisms to resolve direct message policies, manage allowlists for incoming messages, and handle group access. The BlueBubbles plugin, for example, enforces `dmPolicy` and `groupPolicy` and manages `allowFrom` lists. Discord defines direct message policies and advises on `groupPolicy` configurations. Feishu provides detailed policy management for groups and allowlists. Google Chat determines DM policies and validates webhook requests to ensure secure communication. The Matrix plugin manages direct messaging policies, allowlists, and per-room configurations, including mention gating. Mattermost also defines `dmPolicy` and handles `allowFrom` lists to regulate access.

Onboarding processes guide users through the initial setup and configuration of each channel plugin. These processes typically involve interactive prompts for credentials, policy settings, and other channel-specific configurations. The BlueBubbles plugin uses `blueBubblesOnboardingAdapter` for interactive configuration of DM policies and allowlists. Discord similarly uses `discordOnboardingAdapter` for its setup. Feishu's `feishuOnboardingAdapter` assists with setting DM and group policies, and testing credentials. Google Chat's `googlechatOnboardingAdapter` prompts for service account credentials and webhook audience. The Matrix plugin's `matrixOnboardingAdapter` supports interactive authentication, DM policies, and group access settings. Mattermost's `mattermostOnboardingAdapter` helps users configure bot tokens and base URLs.

Runtime monitoring ensures the continuous operation and health of channel integrations. This includes initiating WebSocket connections, processing real-time events, and providing status information. The BlueBubbles plugin monitors inbound webhooks, authenticates requests, and dispatches messages to the core OpenClaw system. Discord's runtime monitors for events and processes them. Feishu manages WebSocket connections and event processing through its `monitorFeishuProvider`. Google Chat processes incoming webhooks and dispatches replies. The iMessage plugin manages the provider process and collects status issues. LINE monitors account lifecycles and handles message delivery. The Matrix extension actively monitors client activity, registers event listeners, and processes messages. Mattermost establishes WebSocket connections to process inbound messages and manages media attachments. Further details on daemon services, which underpin much of this monitoring, can be found in [Daemon Service Management](#).

Authentication Provider Plugins


User
Initiates Auth
OpenClaw Plugin (Antigravity/Gemini/MiniMax)
PKCE Challenge & Auth Request
Exchange Code for Tokens
Returns Access/Refresh Tokens
Authenticates & Authorizes
Presents Login / Verification URL
External OAuth Service (Google/MiniMax)
Redirect w/ Auth Code
Redirect to Login
Browser / Local Server (for Callback)

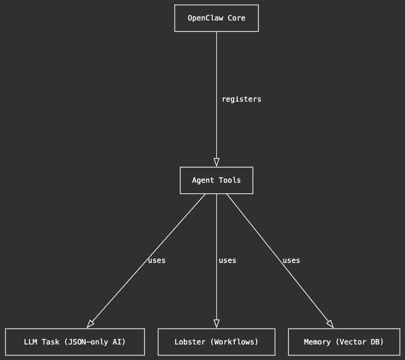
The OpenClaw ecosystem integrates with various external services through OAuth provider plugins, managing secure authentication flows, token exchange, and credential discovery. These plugins facilitate both automatic and manual authentication processes, supporting services such as Google Antigravity (Cloud Code Assist), Google Gemini CLI, and MiniMax.

For Google Antigravity (Cloud Code Assist), the plugin manages the OAuth 2.0 authentication process, including the Proof Key for Code Exchange (PKCE) flow. It handles authorization URL construction, sets up a temporary local HTTP server for automatic redirects, and provides a manual fallback for remote environments or Windows Subsystem for Linux (WSL2). Token exchange is managed by exchanging authorization codes for access and refresh tokens, followed by the retrieval of user email and Google Cloud project ID. This functionality is primarily orchestrated by the `loginAntigravity` function in `extensions/google-antigravity-auth/index.ts`, which also defines the `antigravityPlugin` for OpenClaw integration.

Similarly, the plugin for Google Gemini CLI (Google Code Assist) enables authentication via an OAuth 2.0 flow. It prioritizes credential discovery from environment variables but can also extract client IDs and secrets from an installed `gemini-cli`'s `oauth2.js` file. The `loginGeminiCliOAuth` function in `extensions/google-gemini-cli-auth/oauth.ts` orchestrates the PKCE-based authorization, supporting both local callback mechanisms and manual flows where users are prompted to interact with authorization URLs. Upon successful authentication, it retrieves user email and attempts to discover the associated Google Cloud project ID, using the `CODE_ASSIST_ENDPOINT` for API interactions.

The MiniMax portal authentication plugin supports both global and China-specific endpoints. It implements an OAuth 2.0 device code flow with PKCE, managing the generation of code verifiers and challenges, requesting authorization codes, and continuously polling the token endpoint until user approval. The `loginMiniMaxPortalOAuth` function in `extensions/minimax-portal-auth/oauth.ts` orchestrates this process, guiding the user through the authorization via external functions for opening URLs and providing progress updates. After successful authentication, the `minimaxPortalPlugin` in `extensions/minimax-portal-auth/index.ts` dynamically patches the OpenClaw configuration to include MiniMax models and their associated credentials, accommodating region-specific base URLs and supporting the "Coding plan."

Agent Tool Plugins



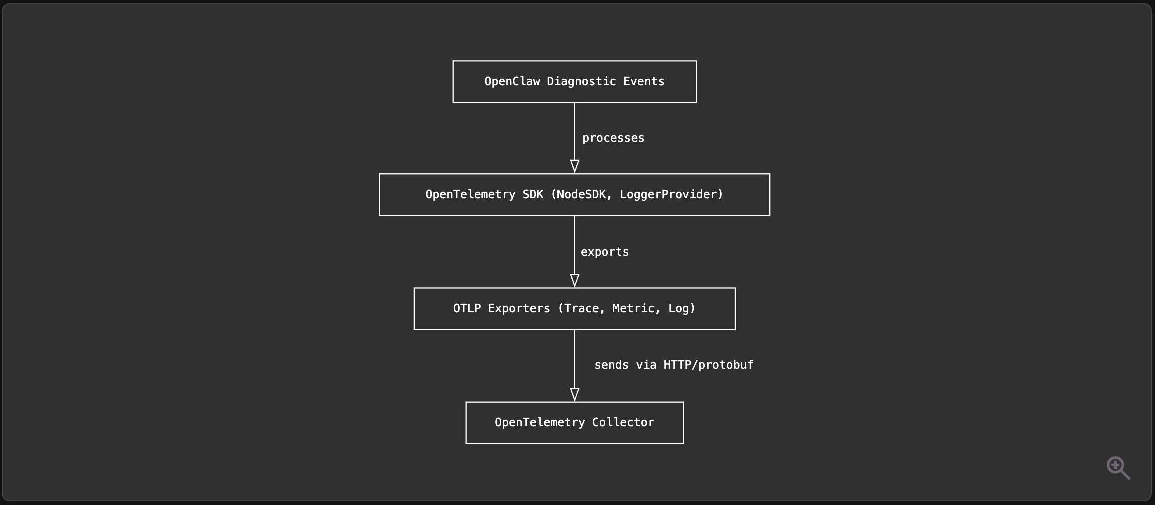
OpenClaw utilizes a plugin architecture to extend its capabilities through various agent tools that automate tasks. These tools integrate specialized functionalities, ranging from executing LLM-based tasks and structured workflows to managing an agent's memory.

The `llm-task` tool, defined within the directory `extensions/llm-task`, enables agents to execute tasks that require Large Language Models (LLMs) to produce JSON-only output. This tool is designed for integration with workflow engines and supports optional JSON Schema validation for the LLM's output. It allows an agent to provide a `prompt` and optional `input` to an LLM, and configure parameters such as the `provider`, `model`, `authProfileId`, and `maxTokens`. The tool ensures that the LLM output is strictly JSON and can validate it against a provided schema. This prevents arbitrary code execution by ensuring the LLM does not have access to other tools within the OpenClaw environment.

For executing and resuming structured workflows, OpenClaw integrates the `lobster` tool, located in `extensions/lobster`. `lobster` is a standalone workflow shell that processes typed, JSON-first pipelines, including support for human approval checkpoints and resume capabilities. An agent can initiate a `lobster` pipeline, and if approval is required, the workflow can be paused and later resumed with an explicit approval. This tool is designed for repeatable automation and actions requiring human intervention, such as email triage or scheduled tasks. Security is a primary concern, with strict validation of executable paths and working directories to prevent unauthorized operations, as detailed in the `SKILL.md` file in `extensions/lobster/SKILL.md`.

Memory management for agents is handled by two distinct tools: `memory-core` and `memory-lancedb`. The `memory-core` plugin, found in `extensions/memory-core`, provides foundational memory functionalities. It registers tools like `memory_search` and `memory_get` for retrieving short-term, file-backed memory, and integrates CLI commands for memory management. Building upon this, the `memory-lancedb` plugin, located in `extensions/memory-lancedb`, provides long-term memory with vector search capabilities. It uses LanceDB for efficient storage of vector embeddings and OpenAI for generating these embeddings. This tool enables agents to `memory_recall` relevant past information, `memory_store` new facts, and `memory_forget` specific memories. It also features `autoCapture` and `autoRecall` lifecycle hooks, allowing the system to automatically store important conversation snippets and recall relevant memories at the beginning of a new agent conversation. The configuration for this long-term memory, including embedding models and database paths, is defined in `extensions/memory-lancedb/config.ts`.

Diagnostics and Telemetry


OpenClaw Diagnostic Events
processes
OpenTelemetry SDK (NodeSDK, LoggerProvider)
exports
OTLP Exporters (Trace, Metric, Log)
sends via HTTP/protobuf
OpenTelemetry Collector

The OpenClaw project integrates OpenTelemetry (OTel) to provide comprehensive diagnostics, enabling the export of operational data as traces, metrics, and logs to an OTel collector. This functionality is managed by an OpenClaw plugin found in the `extensions/diagnostics-otel` directory, which establishes and orchestrates the OpenTelemetry SDK.

At its core, the system initializes `NodeSDK` for capturing traces and metrics, and a `LoggerProvider` for handling logs. This setup uses HTTP/protobuf for data transmission via `OTLPTraceExporter`, `OTLPMetricExporter`, and `OTLPLogExporter`, ensuring compatibility with standard OTel collectors. All telemetry data is associated with a `SemanticResourceAttributes.SERVICE_NAME` attribute, providing context for the collected diagnostics.

The tracing mechanism is configured with a `traceExporter` and can employ `ParentBasedSampler` or `TraceIdRatioBasedSampler` to manage the volume of trace data based on a configurable `sampleRate`. Traces are generated for various OpenClaw events, such as `openclaw.model.usage` and `openclaw.webhook.processed`, with `tracer.startSpan` used to create detailed spans that capture custom attributes and indicate error states.

For metrics, a `PeriodicExportingMetricReader` and `metricExporter` are utilized, with a configurable `exportIntervalMillis`. The system defines specific OpenTelemetry `Counter` and histogram metrics, such as `tokensCounter` and `durationHistogram`, to monitor OpenClaw-specific activities like model token usage, operational costs, and webhook processing.

The logging integration allows OpenClaw's internal log objects to be intercepted and transformed into OpenTelemetry log records. This process maps OpenClaw log levels to OpenTelemetry's `SeverityNumber` and extracts contextual attributes like the logger's name and code location (file path, line number, function).

The lifecycle of the diagnostics service, including the initialization of OTel SDKs and the registration of event listeners, is managed by the `start` method. Conversely, the `stop` method ensures proper resource release by shutting down the SDKs, unregistering diagnostic event listeners, and stopping log transports. This dynamic configuration allows traces, metrics, and logs to be enabled or disabled based on specific operational requirements.

Agent Skills and Tooling

Skill Name	Category	Integrated Tool/API
1password	Secret Management	1Password CLI (op)
apple-notes	Note-taking	`memo` CLI
apple-reminders	Productivity	`remindctl` CLI
bear-notes	Note-taking	`grizzly` CLI
bird	Social Media	`bird` CLI (X/Twitter GraphQL)
blucli	Home Automation	`blu` CLI (Bluesound/NAD)
bluebubbles	Communication	BlueBubbles API (iMessage)
blogwatcher	Content Monitoring	`blogwatcher` CLI (RSS/Atom)
discord	Communication & Collaboration	Discord API
eightctl	Home Automation	`eightctl` CLI (Eight Sleep)
food-order	Food Delivery	`ordercli` CLI (Foodora)
github	Developer Tools	`gh` CLI
gog	Google Workspace	`gog` CLI (Gmail, Calendar, Drive, etc.)
goplaces	Location Services	`goplaces` CLI (Google Places API)
healthcheck	System Security & Maintenance	OpenClaw security tools + OS utilities

The `openclaw` agent framework manages a diverse set of capabilities through a system of "skills," enabling interactions with various external tools and services. These skills are designed to be modular, each providing a specific interface for functionality such as secret management, note-taking, communication, media control, home automation, and productivity tools. Each skill typically resides within its own directory under `skills`, containing metadata and documentation that outline its purpose, installation, configuration, and usage.

A common pattern across these skills is the reliance on external command-line interface (CLI) tools or APIs. For instance, the 1Password skill, detailed in `skills/1password`, integrates with the 1Password CLI (`op`) for secure secret management, emphasizing the use of `tmux` for consistent execution in automated environments. Similarly, the Apple Notes skill (`skills/apple-notes`) and Apple Reminders skill (`skills/apple-reminders`) leverage macOS-specific CLIs (`memo` and `remindctl` respectively) to manage personal information. The Bear notes skill (`skills/bear-notes`) also uses a CLI tool, `grizzly`, to interact with the Bear application's API.

Communication-focused skills extend `openclaw`'s reach into various platforms. The X/Twitter interaction skill (`skills/bird`) provides a CLI for reading, searching, posting, and engagement on the X platform, utilizing cookie-based authentication. For iMessage integration, the BlueBubbles skill (`skills/bluebubbles`) facilitates sending, reacting to, editing, unsending, and replying to messages and attachments through the BlueBubbles platform. Additionally, the Discord skill (`skills/discord`) offers a comprehensive interface for managing messages, reactions, stickers, emojis, polls, threads, and pins within Discord.

Media and content management are also covered. The `blogwatcher` skill (`skills/blogwatcher`) monitors blogs and RSS/Atom feeds, while `gifgrep` (`skills/gifgrep`) enables searching, downloading, and extracting images from GIF providers.

Home automation is addressed through skills like `blucli` (`skills/blucli`), which interacts with BluOS devices for playback, grouping, and volume control, and `eightctl` (`skills/eightctl`), which manages Eight Sleep pods for status, temperature, alarms, and schedules.

Productivity and workspace integrations include the `gog` skill (`skills/gog`) for Google Workspace services such as Gmail, Calendar, Drive, Contacts, Sheets, and Docs. GitHub interaction is handled by the `github` skill (`skills/github`), which uses the `gh` CLI for issues, pull requests, and CI runs, supporting advanced queries with `gh api` and `jq` for structured JSON output.

Location-based services are integrated through `goplaces` (`skills/goplaces`), which queries the Google Places API for search and resolution. For food ordering, the `food-order` skill (`skills/food-order`) utilizes the `ordercli` tool to manage Foodora reordering and tracking, with an emphasis on safe, confirmed transactions.

A critical aspect of the `openclaw` agent framework on macOS is the `swabble` macOS wake-word hook daemon, located in `Swabble`. This daemon performs local, wake-word-gated voice transcription and shell command dispatch. It uses `SwabbleKit` (`Swabble/Sources/SwabbleKit`) to provide multi-platform utilities for wake-word gating, enabling `openclaw` to respond to voice commands locally without relying on cloud services. The `swabble` daemon continuously monitors audio input for a configurable wake word, transcribes subsequent speech, and executes user-defined commands or "hooks," such as launching specific `openclaw` skills. The `HookExecutor` within `SwabbleCore` handles command execution, injecting job data into the process environment.

Agent skills are typically provisioned with installation instructions within their `SKILL.md` files, often leveraging package managers like Homebrew (`brew`) or Go (`go install`). Authentication for these skills generally involves configuring API keys, tokens, or credentials, either through environment variables or dedicated configuration files. The `healthcheck` skill (`skills/healthcheck`) also plays a role in managing OpenClaw deployments by providing security audits, risk posture assessments, and version status checks, which can be scheduled via Cron Job Management System.

Agent Skill Management and Lifecycle

Skill	Primary Install Method	Auth/Config Approach	Config File Location
1Password	Homebrew	Uses 1Password desktop app integration and `op signin`.	N/A
Apple Notes	Homebrew	macOS Automation access to Notes.app.	N/A
Apple Reminders	Homebrew	macOS Reminders permission.	N/A
Bear Notes	Go (`go install`)	Bear app token (`~/.config/grizzly/token`) for some operations.	`~/.config/grizzly/config.toml`
bird (X/Twitter)	npm / Homebrew	Cookie-based auth via `--auth-token`, `--ct0`, or `--cookie-source` (browser cookies).	`~/.config/bird/config.json5` or `./.birdrc.json5`
blogwatcher	Go (`go install`)	N/A	N/A
blucli	Go (`go install`)	Target device selection via `--device`, `BLU_DEVICE` env var, or config default.	N/A
BlueBubbles	OpenClaw config	OpenClaw gateway config (`channels.bluebubbles` for serverUrl/password/webhookPath).	OpenClaw config file
camsnap	Homebrew	Camera host, user, password added via `camsnap add`.	`~/.config/camsnap/config.yaml`
canvas	OpenClaw config	OpenClaw gateway bind mode determines server host.	`~/.openclaw/openclaw.json`
ClawHub	npm	`clawhub login` for publishing.	`./config/mcporter.json` (via `mcporter` for install)
Coding Agent	npm / Homebrew (bins)	Git repo required for Codex, PTY for interactive agents.	`~/.codex/config.toml` for Codex
Discord	OpenClaw config	Bot token configured for OpenClaw.	OpenClaw config file
eightctl	Go (`go install`)	Email and password via `EIGHTCTL_EMAIL`/`EIGHTCTL_PASSWORD` env vars.	`~/.config/eightctl/config.yaml`
food-order	Go (`go install`)	`ordercli foodora login` via email/password or Chrome session.	`--config` flag or default
Gemini	Homebrew	Interactive login flow when first used.	N/A
gifgrep	Homebrew / Go	`GIPHY_API_KEY` for Giphy, `TENOR_API_KEY` optional.	N/A
GitHub	Homebrew / apt	`gh` CLI authentication.	N/A
gog	Homebrew	OAuth setup via `gog auth credentials` and `gog auth add`.	N/A
goplaces	Homebrew	`GOOGLE_PLACES_API_KEY` environment variable.	N/A (env var)
healthcheck	N/A	Relies on OpenClaw's internal security audits and host system tools.	N/A
himalaya	Homebrew	IMAP/SMTP credentials configured in `config.toml`, often with `pass` or keyring for password storage.	`~/.config/himalaya/config.toml`
imsg	Homebrew	Messages.app signed in, Full Disk Access, and Automation permission for Messages.app.	N/A
local-places	`uv` (Python)	`GOOGLE_PLACES_API_KEY` in `.env` or environment.	`.env` file or environment variables
mcporter	npm	OAuth via `mcporter auth`, or key/value pairs for direct tool calls.	`./config/mcporter.json`
model-usage	Homebrew Cask	Requires CodexBar local cost logs.	N/A (uses CodexBar's internal log storage and CLI output)

The `openclaw` agent skills are managed primarily through the `skills` directory, which houses a collection of tools designed to extend the agent's capabilities by integrating with various external services, applications, and system functionalities. Each skill typically corresponds to a specific command-line interface (CLI) tool or API, enabling the agent to perform a wide range of tasks programmatically.

Installation mechanisms for these skills are defined within each skill's `SKILL.md` file. For instance, many skills rely on package managers like Homebrew (`brew`) for macOS (`skills/1password/SKILL.md`, `skills/apple-reminders/SKILL.md`, `skills/camsnap/SKILL.md`, `skills/gemini/SKILL.md`, `skills/gifgrep/SKILL.md`, `skills/github/SKILL.md`, `skills/goplaces/SKILL.md`, `skills/himalaya/SKILL.md`, `skills/imsg/SKILL.md`), Go (`skills/bear-notes/SKILL.md`, `skills/blogwatcher/SKILL.md`, `skills/blucli/SKILL.md`, `skills/eightctl/SKILL.md`, `skills/food-order/SKILL.md`, `skills/gifgrep/SKILL.md`), or npm (`skills/clawhub/SKILL.md`, `skills/mcporter/SKILL.md`). The `metadata.openclaw` block within `SKILL.md` provides structured information for `openclaw` to automate the provisioning of these tools, including their `id`, `kind`, and `bins` requirements. For cross-platform support, the `github` skill provides installation instructions for both `brew` and `apt` (`skills/github/SKILL.md`).

Authentication processes vary depending on the external service. Many skills use API keys stored as environment variables (e.g., `GIPHY_API_KEY` for `gifgrep` in `skills/gifgrep/SKILL.md`, `GOOGLE_PLACES_API_KEY` for `goplaces` in `skills/goplaces/SKILL.md`) or configuration files (e.g., `~/.config/grizzly/token` for `bear-notes` in `skills/bear-notes/SKILL.md`, `~/.config/himalaya/config.toml` for `himalaya` in `skills/himalaya/SKILL.md`). For services like 1Password, authentication involves integrating with a desktop application and using `tmux` sessions for secure `op` command execution (`skills/1password/SKILL.md`). Google Workspace services (`gog`) use `client_secret.json` for initial configuration and support `OAuth2` for subsequent interactions (`skills/gog/SKILL.md`). The `himalaya` skill also supports various `OAuth2` methods, with credential management via external commands or keyrings (`skills/himalaya/references/configuration.md`).

Configuration patterns often involve a dedicated configuration file in the user's home directory (e.g., `~/.config/camsnap/config.yaml` for `camsnap` in `skills/camsnap/SKILL.md`, `~/.config/eightctl/config.yaml` for `eightctl` in `skills/eightctl/SKILL.md`, `~/.config/bird/config.json5` for `bird` in `skills/bird/SKILL.md`), allowing for customization of default behaviors. Environment variables are commonly used for sensitive information like API keys or to override configuration settings, as seen with `BLU_DEVICE` for `blucli` (`skills/blucli/SKILL.md`) and `CLAWHUB_REGISTRY` for `clawhub` (`skills/clawhub/SKILL.md`).

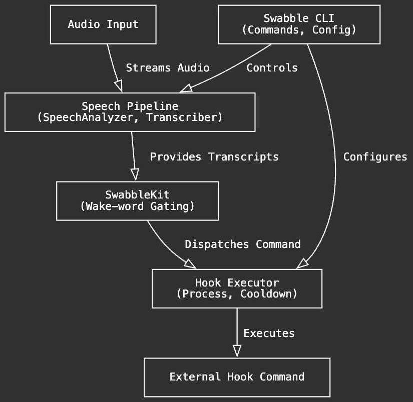
The integration of diverse CLI tools and APIs is a core aspect of skill management. Each skill provides specific actions that map to the underlying tool's capabilities. For example, `apple-notes` leverages `memo` for creating, editing, and searching notes (`skills/apple-notes/SKILL.md`), while `apple-reminders` uses `remindctl` for managing reminders (`skills/apple-reminders/SKILL.md`). Communication channels like `bluebubbles` (`skills/bluebubbles/SKILL.md`), `discord` (`skills/discord/SKILL.md`), and `imsg` (`skills/imsg/SKILL.md`) provide actions for sending messages, managing reactions, and handling attachments. The `canvas` skill enables displaying HTML content on connected nodes through `present`, `hide`, `navigate`, and `eval` actions (`skills/canvas/SKILL.md`).

For coding-related tasks, the `coding-agent` skill orchestrates interactions with various coding agents (e.g., `Codex`, `Claude Code`, `OpenCode`, `Pi`) by executing shell commands with pseudo-terminal (PTY) allocation and managing background processes. It defines patterns for one-shot, long-running, and parallel tasks (`skills/coding-agent/SKILL.md`). The `clawhub` skill facilitates searching, installing, updating, and publishing agent skills from a central repository (`skills/clawhub/SKILL.md`), while `mcporter` provides an interface for interacting with MCP servers and generating client-side code (`skills/mcporter/SKILL.md`).

Other notable integrations include `blucli` for controlling BluOS devices (`skills/blucli/SKILL.md`), `eightctl` for Eight Sleep pods (`skills/eightctl/SKILL.md`), `food-order` for Foodora reordering and tracking (`skills/food-order/SKILL.md`), `gifgrep` for GIF searching and manipulation (`skills/gifgrep/SKILL.md`), `github` for interacting with GitHub issues, pull requests, and CI runs (`skills/github/SKILL.md`), `gog` for Google Workspace services (`skills/gog/SKILL.md`), and `goplaces` and `local-places` for Google Places API queries (`skills/goplaces/SKILL.md`, `skills/local-places/SKILL.md`). The `healthcheck` skill manages host security and OpenClaw deployments, performing audits and providing remediation plans (`skills/healthcheck/SKILL.md`). The `model-usage` skill provides cost summaries for various AI models (`skills/model-usage/SKILL.md`), while `himalaya` offers a terminal-based email client (`skills/himalaya/SKILL.md`). The `peekaboo` skill enables macOS UI automation for system control and vision-based analysis (`skills/peekaboo/SKILL.md`).

A key best practice across these skills is the use of `--json` flags for programmatic output, facilitating easier parsing and integration with other tools. Many skills also prioritize safety, such as the `food-order` skill which requires explicit user confirmation for reordering actions (`skills/food-order/SKILL.md`). The `healthcheck` skill emphasizes explicit approval for state-changing actions and reversible changes (`skills/healthcheck/SKILL.md`). The `coding-agent` skill enforces strict operational guidelines for security and proper procedure, such as using `pty:true` and avoiding sensitive directories (`skills/coding-agent/SKILL.md`).

macOS Wake-Word Gated Voice Transcription (Swabble)


Audio Input
Swabble CLI (Commands, Config)
Streams Audio
Controls
Speech Pipeline (SpeechAnalyzer, Transcriber)
Provides Transcripts
Configures
SwabbleKit (Wake-word Gating)
Dispatches Command
Hook Executor (Process, Cooldown)
Executes
External Hook Command

The `swabble` macOS wake-word hook daemon provides local, on-device voice transcription and shell command dispatch, emphasizing privacy by leveraging Apple's `Speech.framework` for all audio processing without requiring network access. It continuously monitors audio input for a configurable wake word, such as "clawd," and upon detection, transcribes the subsequent speech. The transcribed text is then used to trigger a user-defined shell command or "hook."

The core functionality of `swabble` involves:

- **Wake-Word Detection**: The system listens for specific keywords, defined in the configuration, to initiate transcription and command processing.

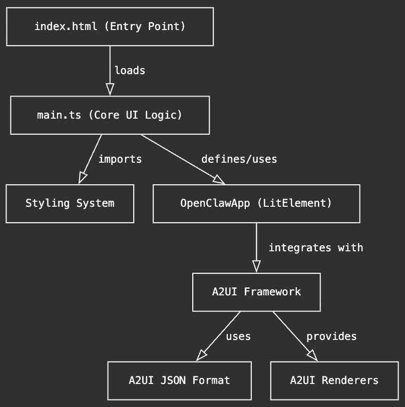
- **Speech Transcription**: Once a wake word is detected, `swabble` transcribes the following speech into text using local macOS capabilities.
- **Command Dispatch**: The transcribed text is passed to a configurable shell command. This allows users to integrate `swabble` with various automation scripts or applications, making the system extensible.
- **Configuration Management**: Settings such as the wake word, audio devices, logging preferences, and the specific command to be executed are managed through a `config.json` file. This configuration is handled by `SwabbleConfig` and `ConfigLoader` in `Swabble/Sources/SwabbleCore/Config/Config.swift`.

The application's architecture includes a `SpeechPipeline` in `Swabble/Sources/SwabbleCore/Speech/SpeechPipeline.swift` that orchestrates audio capture and speech-to-text conversion. The `HookExecutor` in `Swabble/Sources/SwabbleCore/Hooks/HookExecutor.swift` manages the execution of external commands, including cooldowns and environment variable injection for the transcribed text.

For multi-platform integration, `SwabbleKit` in `Swabble/Sources/SwabbleKit/WakeWordGate.swift` provides utilities for wake-word gating that can be incorporated into other iOS and macOS applications. This shared library offers functionality for detecting wake words and extracting commands from speech segments, enforcing minimum gaps between the wake word and the command.

The `swabble` command-line interface (CLI) in `Swabble/Sources/swabble` offers various commands for managing the daemon, including `serve` to run the application, `transcribe` for offline audio transcription, `setup` to initialize the configuration, and `mic` to manage audio input devices. This CLI structure is defined by `CLIRegistry` in `Swabble/Sources/swabble/CLI/CLIRegistry.swift`.

User Interface and Presentation



The OpenClaw project features a web application that serves as its primary user interface, enabling agents to interact with users through dynamic and updateable displays. The application's architecture is designed around clear separation of concerns, encompassing a core UI framework, a robust styling system, and a specialized framework for agent-generated interfaces.

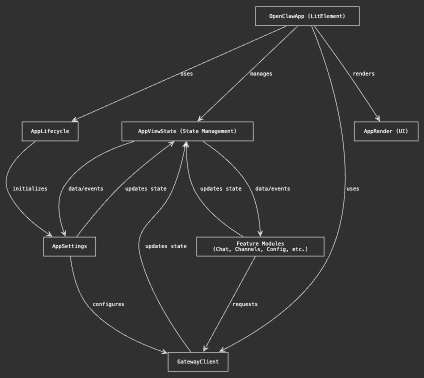
The web application's entry point is defined by the `ui/index.html` file, which sets up the basic HTML structure, metadata, and loads the main JavaScript module from `ui/src/main.ts`. This JavaScript module, in turn, orchestrates the loading of all necessary styles and core application logic.

The core UI logic for the application resides within the `ui/src` directory. This includes the main `OpenClawApp` class in `ui/src/ui/app.ts`, which extends `LitElement` and manages the application's overall state, including user settings, connection status, active tabs, and theme. Key aspects of the application's lifecycle, such as handling connections, updates, and data polling, are managed by modules like `ui/src/ui/app-lifecycle.ts`, `ui/src/ui/app-gateway.ts`, and `ui/src/ui/app-polling.ts`. User interface rendering is dynamically handled by `ui/src/ui/app-render.ts`, which leverages `lit` to present the UI based on the current application state. The application's configuration for build and development is managed by `ui/vite.config.ts`.

A comprehensive styling and theming system is implemented within the `ui/src/styles` directory. This system establishes a consistent visual theme with dark and light modes, defines typography, spacing, and animations, and ensures responsive layouts for various screen sizes. CSS custom properties are extensively used for flexible theming. This includes specific styling for chat interfaces in `ui/src/styles/chat` and configuration pages in `ui/src/styles/config.css`.

A pivotal component of the OpenClaw user interface is the A2UI (Agent-to-User Interface) framework, detailed in `vendor/a2ui`. A2UI defines an open standard for agents to generate and populate rich, updateable user interfaces using a declarative JSON format. This approach separates UI intent from implementation, allowing agents to describe the UI they need without specifying how it should be rendered. This design prioritizes security by relying on client-side catalogs of trusted components, ensures LLM-friendliness by supporting incremental updates to UI structures, and promotes framework-agnostic portability. The framework includes formal specifications for different versions (v0.8 and v0.9) using JSON Schema, and provides evaluation tools for assessing LLM-generated UI components. Concrete renderer implementations for A2UI are available for various frameworks, such as Angular and Lit, located in `vendor/a2ui/renderers`. These renderers process the declarative JSON from agents and map it to native UI components, handling data binding and events while addressing critical security considerations for untrusted data.

Web Application Core Architecture and State Management



The OpenClaw web application is built on a foundational architecture that uses `LitElement` for its component model and a centralized `AppViewState` for managing application state. This structure facilitates a reactive user interface that responds to changes in the underlying data and user interactions.

The entry point for the web application is defined in `ui/index.html`, which loads the main JavaScript module from `ui/src/main.ts`. This module then initializes the core application logic and styling. Global CSS styles are managed through `ui/src/styles.css`, which aggregates modular stylesheets covering base themes, layout, and component-specific styles.

At the heart of the application's functionality is the `OpenClawApp` class, found in `ui/src/ui/app.ts`. This class extends `LitElement` and serves as the root custom element, orchestrating the application's state, connectivity, and rendering. The `AppViewState` type, defined in `ui/src/ui/app-view-state.ts`, provides a comprehensive structure for the application's state, acting as a single source of truth for UI components. Default values for various settings and form states are provided by `ui/src/ui/app-defaults.ts`.

The application's lifecycle, including initialization, updates, and cleanup, is managed by functions in `ui/src/ui/app-lifecycle.ts`. These functions coordinate essential services such as connecting to the OpenClaw gateway, managing data polling, and synchronizing settings. Connectivity to the gateway is handled by `ui/src/ui/app-gateway.ts`, which processes events from the gateway and updates the application's state accordingly. Periodic data fetching for various application sections, including nodes, logs, and debug information, is handled by `ui/src/ui/app-polling.ts` using `setInterval`.

User interface rendering is driven by `ui/src/ui/app-render.ts`, which uses `lit` to dynamically render the UI based on the `AppViewState`. This includes displaying different views depending on the active tab and managing data flow. Utility functions for rendering common UI elements like navigation tabs, chat controls, and theme toggles are provided by `ui/src/ui/app-render.helpers.ts`.

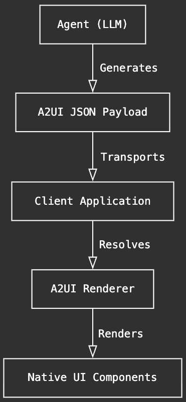
Settings and navigation within the application are managed by `ui/src/ui/app-settings.ts`. This module handles the persistence of settings, applies themes, manages tab navigation, and refreshes data based on the active tab. It also interacts with browser history and URL parameters.

Beyond the core architecture, specific functionalities are compartmentalized into dedicated modules within the `ui/src/ui` directory:

- Chat interface state and interactions are managed by `ui/src/ui/app-chat.ts`, including message sending, queuing, and handling chat control commands.
- Agent tool call events are processed and displayed by `ui/src/ui/app-tool-stream.ts`.
- Scrolling behavior for chat and log areas is managed by `ui/src/ui/app-scroll.ts`.
- User interface interactions and state for WhatsApp and Nostr channel configurations and profile management are handled by `ui/src/ui/app-channels.ts`.
- Assistant identity information is managed by `ui/src/ui/assistant-identity.ts`.
- Device authentication and cryptographic identity are managed by `ui/src/ui/device-auth.ts` and `ui/src/ui/device-identity.ts`, respectively.

The `ui/src/ui/controllers` directory houses modules that abstract interactions with the `GatewayBrowserClient`, managing specific domains such as agents, channels, chat history, configuration, cron jobs, debugging, devices, execution approvals, logs, nodes, presence, sessions, skills, and usage data. Each controller ensures UI synchronization with backend services, handling loading indicators, error management, and reactive state updates. For instance, `ui/src/ui/controllers/config.ts` manages the loading, saving, and updating of application configurations.

A2UI Framework for Agent-Generated UIs



The A2UI (Agent-to-User Interface) framework enables AI agents to generate and dynamically update rich user interfaces. It accomplishes this by using a declarative JSON format to represent UI intent. Client applications, rather than directly executing agent-provided code, receive this JSON payload and map it to their native UI component libraries, ensuring a separation between UI generation and its execution.

A core principle of A2UI is its security-first design. The framework treats the declarative JSON as data rather than executable code. Client applications maintain a catalog of pre-approved UI components, and agents are restricted to requesting components only from this trusted catalog. This approach helps mitigate security risks such as prompt injection, phishing, cross-site scripting (XSS), and denial-of-service (DoS) attacks that could arise from untrusted agent-generated content. Developers are responsible for implementing additional security measures like input sanitization, Content Security Policies (CSP), and isolation of embedded content. The security considerations are detailed in `vendor/a2ui/renderers/angular/README.md` and `vendor/a2ui/renderers/lit/README.md`.

A2UI is designed to be LLM-friendly and support incremental updates. By representing the UI as a flat list of components with ID references, the framework simplifies the task for Large Language Models (LLMs) to generate UI descriptions. This structure also facilitates incremental changes, allowing for progressive rendering and responsive user experiences, as agents can efficiently update only the necessary parts of the UI.

The framework promotes framework-agnostic portability. The agent provides an abstract description of the UI, which client applications then map to their specific native widgets, whether they are web components, Flutter widgets, React components, or SwiftUI views. This abstraction decouples the UI's structure from its implementation.

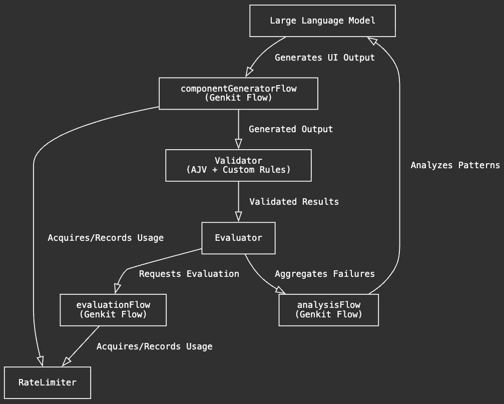
Flexibility is achieved through an open registry pattern that allows developers to associate server-side component types with custom client-side implementations. This includes the ability to integrate "Smart Wrappers" that connect existing UI components, potentially within secure iframe containers, to A2UI's data binding and event system. This mechanism enables the enforcement of sandboxing policies and "trust ladders" for various UI elements.

The overall A2UI flow involves a four-step process:

1. **Generation**: An agent generates an A2UI Response, which is a JSON payload describing the UI component composition and properties.
2. **Transport**: This message is then sent to the client application using established protocols such as the A2A Protocol or AG UI.
3. **Resolution**: The client's A2UI Renderer parses the received JSON payload.
4. **Rendering**: The Renderer translates the abstract component descriptions (e.g., a `type: 'text-field'`) into concrete, native UI implementations within the client's codebase.

The fundamental components and architecture of A2UI are outlined in `vendor/a2ui/README.md`. The framework supports various renderers, including implementations for Angular and Lit, which process A2UI messages to dynamically render UI components and manage data and events. Further details on the specification and evaluation can be found in A2UI Specification and Evaluation Framework, and specifics on rendering implementations are in A2UI Renderer Implementations (Angular and Lit).

A2UI Specification and Evaluation Framework



The A2UI (Agent-to-User Interface) framework enables AI agents to generate and update rich user interfaces. This framework separates UI generation from client-side rendering, emphasizing security, LLM-friendliness, incremental updates, and framework-agnostic portability. It defines formal specifications for its declarative JSON format and provides tools for evaluating how effectively Large Language Models (LLMs) can generate UI components according to these specifications.

The A2UI specification is formally defined through JSON Schema files, located in directories such as `vendor/a2ui/specification/0.8/json` and `vendor/a2ui/specification/0.9/json`. These schemas define the structure of protocol messages that pass between the agent (server) and the client (user application). These messages include instructions for initiating UI rendering, updating UI surfaces, modifying data models, and deleting surfaces. The schemas also describe the structure of A2UI component catalogs, which enumerate the available UI components and their properties. For example, `server_to_client.json` outlines server-to-client messages, while `standard_catalog_definition.json` provides a concrete example of a component catalog. The use of JSON Schema provides a machine-readable contract for UI generation and interpretation.

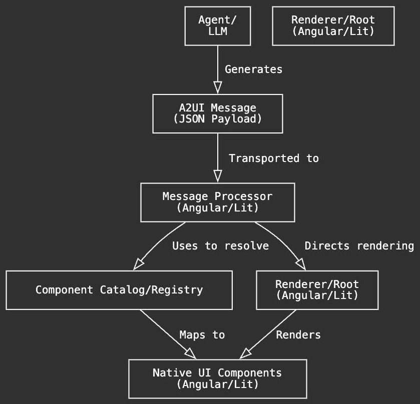
To ensure that LLMs can accurately generate UI that conforms to the A2UI specification, a comprehensive evaluation framework is provided. This framework, found in directories such as `vendor/a2ui/specification/0.8/eval` and `vendor/a2ui/specification/0.9/eval`, assesses LLM performance in generating structured UI component data from natural language prompts. It facilitates the setup of AI runtimes, defines test scenarios, executes generation flows, validates the outputs against the A2UI JSON schemas, and generates detailed reports.

The evaluation process typically involves:

- **Setup**: Configuring access to AI models and initializing the Genkit framework.
- **Generation Flows**: LLMs are prompted to generate UI components based on specific scenarios. These prompts often embed JSON schemas directly, guiding the LLM to produce JSON output within markdown code blocks, as orchestrated by flows like `componentGeneratorFlow` in `vendor/a2ui/specification/0.9/eval/src/generation_flow.ts`.
- **Validation**: The generated JSON output is extracted and rigorously validated against the A2UI JSON schemas and additional custom rules to check for structural correctness, required properties, and inter-component relationships. The `Validator` class in `vendor/a2ui/specification/0.9/eval/src/validator.ts` plays a central role here, leveraging `ajv` for schema validation and performing custom checks for message types like `createSurface` and `updateComponents`.
- **Rate Limiting**: To manage API usage and prevent service disruptions during evaluations, a `RateLimiter` component (e.g., in `vendor/a2ui/specification/0.9/eval/src/rateLimiter.ts`) enforces limits on requests and tokens per minute and includes circuit breaking mechanisms to handle resource exhaustion.
- **Reporting**: After generation and validation, an `Evaluator` (e.g., `vendor/a2ui/specification/0.9/eval/src/evaluator.ts`) assesses the quality of the generated UI, often using another LLM as a QA evaluator, and produces a summary report detailing the performance and any identified issues. An `analysisFlow` can further investigate evaluation failures.

The framework also includes component rules, such as those in `vendor/a2ui/specification/0.9/json/standard_catalog_rules.txt`, which define mandatory properties for standard UI components like `Text`, `Image`, and `Button`. This ensures consistency and functionality across generated UIs. Furthermore, utility scripts like `vendor/a2ui/specification/0.9/validate.sh` are provided for validating JSONL data against the defined schemas, reinforcing the robustness of the A2UI ecosystem.

A2UI Renderer Implementations (Angular and Lit)


Agent/ LLM
Renderer/Root (Angular/Lit)
Generates
A2UI Message (JSON Payload)
Transported to
Message Processor (Angular/Lit)
Uses to resolve
Directs rendering
Component Catalog/Registry
Renderer/Root (Angular/Lit)
Maps to
Renders
Native UI Components (Angular/Lit)

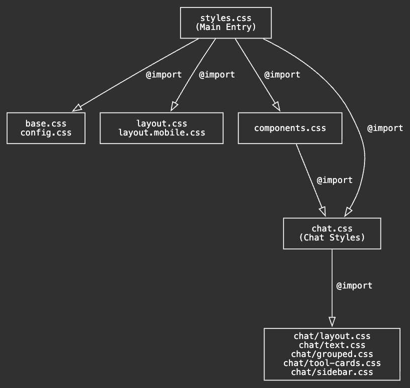
The A2UI framework provides concrete renderer implementations for Angular and Lit, enabling client applications to dynamically render user interfaces described by agents. These renderers process A2UI messages to display UI components, manage data flow, and handle user interactions, all while adhering to critical security considerations due to the untrusted nature of agent-generated data. For a general overview of the A2UI framework, refer to A2UI Framework for Agent-Generated UIs.

The `vendor/a2ui/renderers` directory contains these different renderer implementations. Both the Angular and Lit renderers prioritize security, acknowledging the risks associated with Agent-to-Agent (A2A) interactions. These risks include prompt injection, phishing attempts, cross-site scripting (XSS), denial of service (DoS) attacks, and exposure to malicious external sites. Developers are responsible for implementing safeguards such as input sanitization, Content Security Policies (CSP), and isolation for embedded content.

The Angular renderer, located in `vendor/a2ui/renderers/angular`, configures the A2UI environment using a component catalog and thematic settings. It includes a `DynamicComponent` base class that provides common UI component functionalities for handling properties and dispatching actions. A `Renderer` directive dynamically renders components based on agent-provided definitions, managing their lifecycle and applying structural styles. The component catalog (`vendor/a2ui/renderers/angular/src/lib/catalog`) maps abstract A2UI node types (like buttons, text fields, or images) to concrete Angular components, facilitating the translation of agent intent into a functional UI. A `MessageProcessor` handles A2UI client events and request-response patterns, while a `MarkdownRenderer` converts Markdown content into sanitized HTML.

Similarly, the Lit renderer, found in `vendor/a2ui/renderers/lit`, demonstrates the A2UI and A2A protocol mechanics using Lit-based web components. It manages A2UI protocol messages and the UI component tree through a reactive data model provided by `A2uiMessageProcessor` (`vendor/a2ui/renderers/lit/src/0.8/data/model-processor.ts`). This processor handles various server-to-client messages (e.g., `beginRendering`, `surfaceUpdate`, `dataModelUpdate`, `deleteSurface`) to update the UI dynamically. Custom events are defined for structured communication of user actions and state changes. The Lit renderer also provides a comprehensive styling system and includes a set of custom web components (`vendor/a2ui/renderers/lit/src/0.8/ui`) that handle data binding and render interactive elements, including a `MarkdownDirective` for safe Markdown conversion. The `ComponentRegistry` (`vendor/a2ui/renderers/lit/src/0.8/ui/component-registry.ts`) manages the registration and retrieval of these custom elements.

Styling and Theming System



The OpenClaw web application utilizes a comprehensive CSS styling architecture designed to ensure a consistent visual experience, support theming, and adapt to various screen sizes. This architecture centralizes styling definitions through CSS custom properties, promoting maintainability and enabling dynamic theme switching.

The foundational styles are defined in `ui/src/styles/base.css`, which establishes a set of variables for colors, typography, spacing, shadows, and animation timings. This file includes definitions for both dark (default) and light themes, allowing a complete inversion of the color scheme while maintaining structural consistency. Global styles for common HTML elements and custom scrollbar styling are also managed here. Keyframe animations for UI effects, such as `theme-circle-transition` for theme switching, `rise` for element entry, and `shimmer` for loading states, are centrally defined in this file.

The application's overall layout is managed by `ui/src/styles/layout.css` and its mobile-specific counterpart, `ui/src/styles/layout.mobile.css`. These files use CSS Grid for structuring the page into a top navigation bar, a left-hand navigation sidebar, and a main content area. Responsive design principles are applied to adapt the layout for various screen sizes, including collapsing the navigation or modifying element visibility on smaller devices.

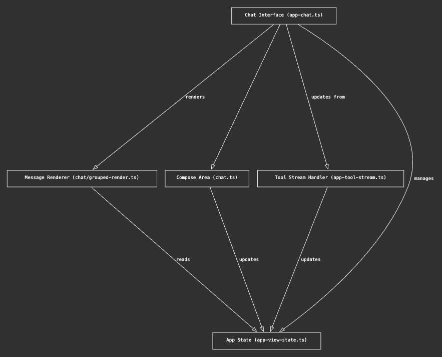
Component-specific styles are grouped in `ui/src/styles/components.css`. This file provides styling for a wide array of UI elements, including cards, statistics, buttons, form elements, lists, and various components within agent management layouts. It leverages CSS variables for theming and incorporates animations for dynamic behavior.

The chat interface has its own dedicated styling within the `ui/src/styles/chat` directory. This includes styles for message grouping, individual message bubbles, avatars, interactive elements, and text formatting, encompassing code blocks and blockquotes. This modular approach to chat styling enables visual consistency across its various features. Responsive adjustments for chat elements and a collapsible sidebar are also integrated here.

For the application's configuration user interface, styles are defined in `ui/src/styles/config.css`. This includes a responsive two-column layout with a sidebar and main content area, comprehensive styling for various form elements (inputs, toggles, segmented controls), and visual indicators for loading or empty states.

Collectively, these styling components ensure a unified and adaptable visual presentation for the OpenClaw web application, facilitating consistent user experience across different features and devices.

Chat Interface Functionality


Chat Interface (app-chat.ts)
renders
updates from
Message Renderer (chat/grouped-render.ts)
Compose Area (chat.ts)
Tool Stream Handler (app-tool-stream.ts)
manages
reads
updates
updates
App State (app-view-state.ts)

The chat interface provides the primary means for users to interact with OpenClaw agents. It centralizes the display and management of messages, tool outputs, and user input. The core functionality manages the visual presentation of conversations, ensuring consistent formatting and dynamic updates.

Message handling involves normalizing incoming chat messages into a consistent structure for display, which includes identifying message roles (user, assistant, tool results) and assigning metadata. Text content is extracted from messages for display, along with internal "thinking" or "reasoning" content from agents. This extraction process can utilize caching to optimize performance. For more general UI styling principles, see Styling and Theming System.

The rendering process organizes messages into logical groups within the chat thread, distinguishing between user input and agent responses. Avatars for different roles (user, assistant, tool) are displayed alongside messages to visually identify the sender. The interface also supports the display of images embedded within messages.

Interactive tool cards are a key feature, representing agent actions or tool outputs. These cards can be rendered in a sidebar, providing detailed views of tool calls and their results, including truncated previews for conciseness. A "reading" indicator provides visual feedback when an agent is processing information. The overall chat layout handles the header, the scrollable message thread, and the message composition area, including attachment previews. For a broader understanding of the web application's architecture, refer to Web Application Core Architecture and State Management.

The chat functionality also includes utilities for managing user input, such as sending messages and queuing them if an agent is busy. Special commands like `/stop` or `/new` can control the chat flow, allowing users to abort ongoing processes or start new conversations. Message streaming is supported, with visual cues like pulsing borders to indicate active generation. Users can copy message content to their clipboard, with visual feedback indicating success or failure. The chat interface coordinates with backend services to load chat history and session data, and to update agent avatars. The overarching application lifecycle coordinates various services, including gateway connections and data polling. See Web Application Core Architecture and State Management for more details.

Development Operations and Utility Scripts

Script Name	Purpose	Category
`auth-monitor.sh`	Monitors Claude Code authentication expiry and sends proactive notifications.	AI Authentication
`bench-model.ts`	Benchmarks AI model performance by running a prompt multiple times and summarizing durations.	Benchmarking
`build-and-run-mac.sh`	Builds and launches the macOS application in debug mode.	macOS Development
`build_icon.sh`	Renders the macOS `.icon` bundle into a padded `.icns` file.	macOS Development
`bundle-a2ui.sh`	Bundles A2UI (Accessibility and AI UI) assets for the Canvas host.	UI Development
`canvas-a2ui-copy.ts`	Copies A2UI assets from source to a distribution directory.	UI Development
`changelog-to-html.sh`	Extracts and converts a specific version's changelog entries from Markdown to HTML.	Documentation
`claude-auth-status.sh`	Checks the authentication status for Claude Code and OpenClaw.	AI Authentication
`clawlog.sh`	Utility for accessing and filtering OpenClaw logs using the macOS unified logging system.	Logging & Debugging
`codesign-mac-app.sh`	Codesigns the macOS application bundle with a specified identity and entitlements.	macOS Development
`copy-hook-metadata.ts`	Copies `HOOK.md` metadata files from source to distribution for bundled hooks.	Hooks Management
`create-dmg.sh`	Creates a styled macOS Disk Image (.dmg) containing the application bundle.	macOS Packaging
`debug-claude-usage.ts`	Debugs Claude AI API usage by checking various authentication token sources.	AI Authentication
`docs-list.js`	Lists Markdown files in the `docs` directory and extracts front matter metadata.	Documentation
`firecrawl-compare.ts`	Compares content extraction using Firecrawl and local readability tools for given URLs.	Web Scraping & Benchmarking

`ios-team-id.sh`	Retrieves the Apple Team ID configured for Xcode.	iOS Development
`make_appcast.sh`	Generates an `appcast.xml` file for Sparkle automatic updates from a ZIP archive.	macOS Packaging
`mobile-reauth.sh`	Provides a mobile-friendly re-authentication flow for Claude Code via `claude setup-token`.	AI Authentication
`notarize-mac-artifact.sh`	Notarizes a macOS artifact (zip, dmg, pkg) and optionally staples the app bundle.	macOS Packaging
`package-mac-app.sh`	Builds and bundles the macOS application into a `.app` directory.	macOS Packaging
`package-mac-dist.sh`	Builds the macOS app bundle, then creates a Sparkle-compatible ZIP and a styled DMG.	macOS Packaging
`protocol-gen-swift.ts`	Generates Swift data models from the Gateway Protocol schemas.	Protocol Generation
`protocol-gen.ts`	Generates a JSON Schema file for the OpenClaw Gateway Protocol.	Protocol Generation
`readability-basic-compare.ts`	Compares basic readability extraction methods for given URLs.	Web Scraping & Benchmarking
`release-check.ts`	Verifies the integrity of files included in the `npm pack` for release.	Release Management
`restart-mac.sh`	Resets and relaunches the macOS application by killing instances, rebuilding, and repackaging.	macOS Development
`sandbox-browser-entrypoint.sh`	Entrypoint for a Dockerized sandbox browser, setting up Xvfb, Chrome, VNC, and noVNC.	Sandboxing & Docker
`sandbox-browser-setup.sh`	Builds the Docker image for the sandbox browser environment.	Sandboxing & Docker
`sandbox-common-setup.sh`	Builds a Docker image with common development tools (Node.js, Go, Rust, etc.) for sandboxes.	Sandboxing & Docker
`sandbox-setup.sh`	Builds the base Docker image for the sandboxing environment.	Sandboxing & Docker
`setup-auth-system.sh`	Sets up a long-lived Claude Code token, auth monitoring, and Termux widget instructions.	AI Authentication
`sync-labels.ts`	Synchronizes GitHub issue labels from `labeler.yml` to the repository.	GitHub Automation
`sync-moonshot-docs.ts`	Synchronizes Moonshot model information into documentation files.	Documentation
`sync-plugin-versions.ts`	Synchronizes plugin versions in `package.json` files to match the root package version.	Plugin Management
`termux-auth-widget.sh`	Termux widget to check OpenClaw auth status and guide re-authentication on a phone.	Mobile Utilities
`termux-quick-auth.sh`	Minimal Termux widget for a quick auth check; opens auth URL if expired.	Mobile Utilities
`termux-sync-widget.sh`	Termux widget to sync Claude Code tokens to OpenClaw on a remote server.	Mobile Utilities
`test-cleanup-docker.sh`	Builds and runs a Docker image to smoke test the `openclaw reset` and `uninstall` commands.	Testing & Docker
`test-force.ts`	Forces termination of processes listening on a specified port, then runs Vitest tests.	Testing & Debugging
`test-install-sh-docker.sh`	Builds and runs Docker images to test the `install.sh` script for root and non-root users.	Testing & Docker
`test-install-sh-e2e-docker.sh`	Builds and runs a Docker image for end-to-end testing of the `install.sh` script with AI models.	Testing & Docker
`test-live-gateway-models-docker.sh`	Runs live AI model tests against the gateway within a Docker container.	Testing & Docker
`test-live-models-docker.sh`	Runs live AI model tests within a Docker container, using configured profile keys.	Testing & Docker
`test-shell-completion.ts`	Simulates shell completion installation prompts for testing purposes.	Testing
`ui.js`	Generic script to manage UI tasks (install, dev, build, test) using `pnpm`.	UI Development
`update-clawtributors.ts`	Updates the "clawtributors" section of the `README.md` based on GitHub contributions.	GitHub Automation
`write-build-info.ts`	Writes build information (version, commit, timestamp) to `dist/build-info.json`.	Build & Release
`write-cli-compat.ts`	Writes a legacy compatibility shim for CLI daemon imports.	Build & Release
`zai-fallback-repro.ts`	Reproduces a Z.AI fallback scenario for model agents in a test environment.	Testing & AI Models
`docs-i18n/` (Go scripts)	Tools for internationalizing documentation, including translation memory, glossary loading, and HTML/Markdown processing.	Documentation
`e2e/` (Docker scripts)	End-to-end testing scripts for various OpenClaw functionalities like doctor, gateway networking, onboarding, and plugins.	Testing & Docker
`pre-commit/run-node-tool.sh`	A utility to run Node.js tools (e.g., linters) using the preferred package manager.	Development Utilities
`repro/tsx-name-repro.ts`	A minimal script to reproduce `tsx` module name loading issues.	Testing
`docker/` (Dockerfile & run.sh)	Dockerfiles and run scripts for various testing and sandbox environments.	Docker & Testing

The OpenClaw project leverages a suite of scripts for managing various development and operational tasks. These scripts streamline processes from application development and deployment to monitoring and documentation.

A key set of utilities focuses on macOS application development and distribution. This includes scripts such as `scripts/build-and-run-mac.sh` for building and launching the `OpenClaw` macOS application for development purposes. Icon generation for macOS is handled by `scripts/build_icon.sh`, which converts a source `.icon` file into the `.icns` format required by the operating system. The process of signing macOS applications, including identity selection, entitlement generation, and recursive signing of embedded components, is automated by `scripts/codesign-mac-app.sh`. The final packaging and distribution to users are supported by `scripts/create-dmg.sh`, which creates styled macOS Disk Images (DMG). The complete workflow for building, packaging, notarizing, and distributing the macOS application is orchestrated by a set of scripts described in macOS Application Build, Notarization, and Distribution.

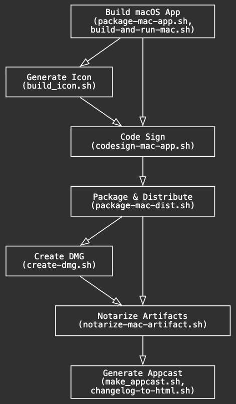
For managing AI-related authentication, scripts like `scripts/auth-monitor.sh` are in place to monitor the expiration of Claude Code authentication credentials, providing proactive notifications before they expire. Further diagnostics for Claude AI token usage are handled by `scripts/debug-claude-usage.ts`, which retrieves and validates access tokens from various sources and fetches usage data from both the Anthropic API and the Claude AI web interface. More details on these monitoring and diagnostic tools can be found in Claude AI Authentication Monitoring and Usage Diagnostics.

Benchmarking language models is facilitated by `scripts/bench-model.ts`, which measures response times and token usage over multiple runs for a given prompt, allowing for performance comparisons between different models.

Documentation and UI-related scripts include `scripts/bundle-a2ui.sh`, which bundles A2UI (Ambient Aether User Interface) components, and `scripts/changelog-to-html.sh`, which extracts changelog entries from Markdown and converts them to HTML format, useful for generating web-friendly release notes. For TypeScript and TSX files, `scripts/check-ts-max-loc.ts` helps maintain code quality by checking file sizes against a maximum line count and reporting files that exceed the limit. The `clawlog` utility, located at `scripts/clawlog.sh`, provides advanced logging capabilities specifically for OpenClaw on macOS, allowing for filtering and display of relevant logs, including handling `sudo` requirements for full log access.

The project also includes Dockerized environments for comprehensive end-to-end testing of the `openclaw` CLI, covering installation, cleanup, and functional verification across different scenarios. These environments are detailed in Dockerized End-to-End Testing Environments. Additionally, documentation internationalization is supported by a Go module located at `scripts/docs-i18n`, which automates the translation of Markdown files using translation memory, glossary data, and markdown masking, as outlined in Documentation Internationalization (Go Module).

macOS Application Build, Notarization, and Distribution



The OpenClaw project includes a comprehensive set of scripts for automating the build, packaging, notarization, and distribution of its macOS application. This workflow is designed to ensure that the application is correctly signed, bundled, and prepared for release, including support for updates via Sparkle and `.dmg` creation.

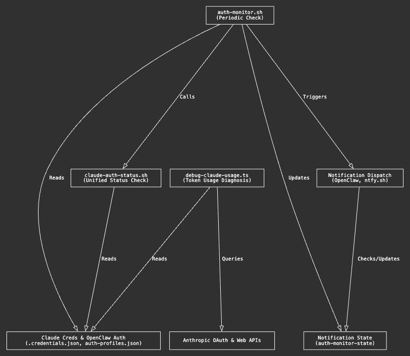
The process begins with building a debug version of the macOS application, which can be accomplished using the script located at `scripts/build-and-run-mac.sh`. This script handles the compilation of the Swift project and launches the application for development purposes. Icon generation for the application is managed by `scripts/build_icon.sh`, which takes a source `.icon` file, processes it, and scales it to various required sizes before compiling it into a `.icns` file suitable for macOS applications.

A critical step in the distribution workflow is code signing, which is automated by the `scripts/codesign-mac-app.sh` script. This script handles the selection of appropriate signing identities, generates necessary entitlements, and applies timestamping to the signed code. It recursively signs embedded components within the application bundle, such as frameworks and libraries, to ensure the integrity and security of the entire package. The script also includes logic to validate Team IDs across all Mach-O files to prevent common Gatekeeper issues.

For distribution, the application is packaged into a `.dmg` (Disk Image) file using the `scripts/create-dmg.sh` script. This script not only creates the `.dmg` but also allows for visual customization, including background images, window bounds, and icon positioning, providing a polished user experience during installation. Before final distribution, the application artifact (e.g., `.dmg` or `.pkg`) can be notarized using `scripts/notarize-mac-artifact.sh`. Notarization is an Apple requirement that checks software for malicious content and verifies developer identity, enhancing user trust. This script submits the artifact to Apple's notarization service and can also staple the notarization ticket to the application.

To support automatic updates, the workflow includes generating an Appcast XML file, which is a standard for Sparkle-based update mechanisms. The `scripts/make_appcast.sh` script generates this XML file, incorporating release notes derived from the project's changelog. The changelog entries themselves can be extracted and converted to HTML format for inclusion in release notes by `scripts/changelog-to-html.sh`. The overall packaging process, which integrates building, optional notarization, `.dmg` creation, and `.dSYM` packaging, is orchestrated by the `scripts/package-mac-dist.sh` script.

Claude AI Authentication Monitoring and Usage Diagnostics

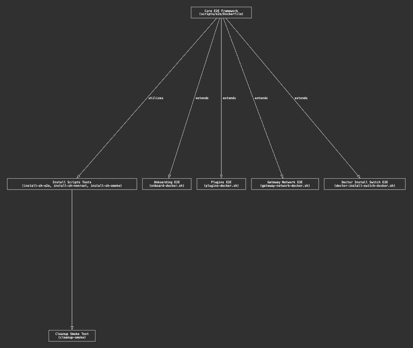


The project includes tools for monitoring the authentication status of Claude AI and diagnosing potential issues with token usage. These tools help ensure continuous operation by providing timely notifications about expiring credentials and offering utilities to inspect and validate access tokens.

Authentication monitoring is handled by a script located at `scripts/auth-monitor.sh`, which periodically checks the expiration status of Claude Code authentication. If credentials are found to be expired or approaching expiration within a configurable warning window, the script can dispatch notifications. This proactive approach helps users re-authenticate before service interruption. This script determines the authentication status by parsing the `expiresAt` timestamp from the `CLAUDE_CREDS` file and comparing it against the current time. It leverages another script, `scripts/claude-auth-status.sh`, to verify the current authentication state before sending notifications. To prevent notification fatigue, a file-based state management system tracks the last notification time, ensuring alerts are not sent too frequently. Notifications can be delivered via various channels, including direct phone messages using an `openclaw` command-line tool or push notifications through `ntfy.sh`.

For diagnosing token usage issues, a dedicated TypeScript script, `scripts/debug-claude-usage.ts`, retrieves and validates Claude AI access tokens from multiple sources. This script aims to provide a comprehensive view of how tokens are managed and consumed. It can load authentication profiles from `auth-profiles.json`, access programmatic tokens stored in the macOS keychain, and even extract session keys from browser cookie databases (Firefox, Chrome, Arc, Brave, Edge). Once tokens are retrieved, the script attempts to fetch usage data directly from the Anthropic API for programmatic tokens and simulates web requests to Claude AI's web interface for session-based usage. This multi-source token retrieval and usage data aggregation assist in pinpointing where a token issue might originate. Sensitive token values are masked in the output by default to enhance security, with an option to reveal them for detailed debugging. The script's design incorporates platform-specific logic, such as macOS keychain access and browser cookie decryption, to cater to different operating environments.

Dockerized End-to-End Testing Environments



The OpenClaw project utilizes a suite of Dockerized environments for comprehensive end-to-end testing of its command-line interface (CLI) and core functionalities. These environments facilitate isolated and reproducible testing of installation, cleanup, configuration, agent onboarding, and tool usage across various user types and upgrade scenarios. The Docker images are built upon `node:22-bookworm` or `node:22-bookworm-slim`, providing consistent Node.js environments.

The system includes Dockerized smoke tests for the `openclaw` CLI's cleanup commands, `reset` and `uninstall`, verifying that configuration, credentials, and session data are thoroughly removed. This is managed by the directory `scripts/docker/cleanup-smoke`.

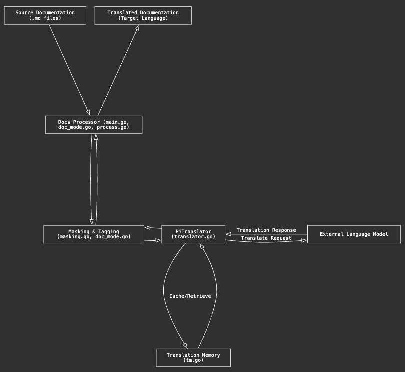
End-to-end installation tests are conducted within Docker to validate the `openclaw` installer and its ability to handle upgrades. These tests cover scenarios with both OpenAI and Anthropic models, ensuring proper configuration and agent interaction. This functionality is supported by the directory `scripts/docker/install-sh-e2e`.

Specific environments are dedicated to verifying installations for non-root users, ensuring the CLI functions correctly with restricted permissions. These tests involve provisioning a non-root user within a Docker container and then executing the installation script. This is handled by the directory `scripts/docker/install-sh-nonroot`.

Further smoke tests for the installation script include simulating upgrade scenarios, checking if the `openclaw` CLI is correctly installed and updated to its latest version. This is found in the directory `scripts/docker/install-sh-smoke`.

The end-to-end testing framework also encompasses tests for the `openclaw` daemon's installation switch functionality, verifying that the service entrypoint can be switched between `npm`-global and `git`-local installations using `daemon install` and `doctor --repair` commands. Gateway network capabilities are also tested, ensuring proper communication between the gateway and client containers. Furthermore, interactive onboarding and configuration wizards are validated, simulating user input to `clack` prompts, and the plugin system's loading, tool registration, and various installation methods (e.g., `.tgz` archives, local directories, `file:` npm specifications) are thoroughly exercised. A smoke test for `qrcode-terminal` import functionality is also included. The core orchestration for these tests resides in the directory `scripts/e2e`, with a foundational Dockerfile at `scripts/e2e/Dockerfile` that prepares the project for testing.

Documentation Internationalization (Go Module)



The Go module for documentation internationalization, located in `scripts/docs-i18n`, automates the translation of documentation files from a source language to a target language. This system is designed to streamline the process of making OpenClaw's documentation available in multiple languages by leveraging machine translation with advanced features.

At its core, the module coordinates translation by parsing documentation files, translating their content, and then reassembling the translated output. It supports two primary modes of operation: `doc` mode, which translates entire documentation files, and `segment` mode, which focuses on translating individual text segments within files. This flexibility allows for different strategies depending on the complexity and structure of the documentation.

To enhance translation quality and efficiency, the system utilizes a translation memory (TM) and glossary data. The TM, managed by functions in `scripts/docs-i18n/tm.go`, stores previously translated content, allowing for the reuse of translations and ensuring consistency. The glossary, loaded via `scripts/docs-i18n/glossary.go`, provides domain-specific terms and their preferred translations, guiding the translation process to maintain accuracy and brand voice.

A key challenge in translating technical documentation is handling code snippets, URLs, and other structured content that should not be altered by the translation engine. This module addresses this through markdown masking, implemented in `scripts/docs-i18n/masking.go`. This process temporarily replaces such elements with placeholders before translation, and then restores them afterward, preserving the integrity of the original markdown structure.

The translation process itself is orchestrated by `PiTranslator`, which interacts with a language model service. This translator, detailed in `scripts/docs-i18n/translator.go`, incorporates retry logic with exponential backoff to handle transient service errors and generates dynamic prompts to instruct the language model on translation rules, style, and the use of the provided glossary.

For efficiency, particularly with large sets of documentation, the `doc` mode supports parallel processing. This allows multiple files to be translated concurrently, speeding up the overall internationalization workflow. The system also includes logic to prevent unnecessary re-translation of unchanged files by comparing source hashes, as outlined in `scripts/docs-i18n/doc_mode.go`.

Utility functions in `scripts/docs-i18n/util.go` support the overall workflow by generating unique cache keys, hashing text content for consistency, and normalizing text for reliable processing. The ordering of files for processing is handled by `scripts/docs-i18n/order.go`, ensuring a predictable and consistent translation sequence.

Issue and Feature Management

```yaml
name: Bug report
about: Report a problem or unexpected behavior in Clawdbot.
title: "[Bug]: "
labels: bug
---

## Summary
What went wrong?

## Steps to reproduce
1. (Truncated for brevity)

## Expected behavior
What did you expect to happen?

## Actual behavior
What actually happened?

## Environment
# ... (rest of the template, including logs/screenshots)
```

The `Clawdbot` project utilizes GitHub issue templates to standardize the process of reporting bugs and requesting features. These templates, located in the `.github/ISSUE_TEMPLATE` directory, ensure that developers receive comprehensive and consistent information, which aids in efficient issue management, prioritization, and resolution. The structured approach helps users provide necessary details for both identifying problems and suggesting enhancements.

Bug Reporting Workflow and Template

Section	Purpose/Details
Summary	Briefly describe what went wrong.
Steps to reproduce	Provide a numbered list of actions to trigger the bug.
Expected behavior	Describe what you anticipated would happen.
Actual behavior	Explain what actually occurred, contrasting with expected behavior.
Environment	Detail the software version, operating system, and installation method.
Logs or screenshots	Include relevant logs or visual aids to support the report.

The `Clawdbot` project utilizes a structured process for reporting bugs, leveraging a predefined GitHub issue template located at `.github/ISSUE_TEMPLATE/bug_report.md`. This template standardizes the information collected from bug reports, ensuring that developers receive comprehensive details necessary for replication, prioritization, and resolution.

When a user reports a bug, they are guided to provide a `Summary` of the problem. Following this, the template requires `Steps to reproduce` the bug, which is crucial for development teams to consistently replicate the issue. Users must also describe the `Expected behavior` versus the `Actual behavior`, clearly highlighting the deviation that constitutes the bug. To assist in diagnosing environment-specific problems, the template requests `Environment` details, including the `Clawdbot version`, operating system (`OS`), and `Install method`. Finally, a dedicated section allows for the inclusion of `Logs or screenshots`, with a cautionary note to redact any sensitive information.

This structured approach ensures that bug reports are consistent and thorough, facilitating an efficient debugging process. For a broader overview of how issues and features are managed, see Issue and Feature Management. In addition to bug reports, a similar templated approach is used for suggesting new features, as described in Feature Request Workflow and Template.

Feature Request Workflow and Template

```markdown
name: Feature request
about: Suggest an idea or improvement for Clawdbot.
title: "[Feature]: "
labels: enhancement

---

## Summary
<!-- Describe the problem you are trying to solve or the opportunity you see. -->

## Proposed solution
<!-- What would you like Clawdbot to do? -->

## Alternatives considered
<!-- Any other approaches you have considered? -->

## Additional context
<!-- Links, screenshots, or related issues. -->
```

The process for submitting feature requests or improvements to the `Clawdbot` project is standardized through a dedicated GitHub issue template, located at `.github/ISSUE_TEMPLATE/feature_request.md`. This template guides users in providing essential information to ensure that feature requests are clear, comprehensive, and actionable.

When a user initiates a new feature request, the template automatically labels the issue as an `enhancement` and prefills the issue title with `[Feature]:`. The core components of a feature request, as outlined by the template, include: a summary of the problem or opportunity the proposed feature addresses; a detailed description of the proposed solution, articulating what `Clawdbot` should achieve; any alternative solutions or approaches that have been considered; and additional context, such as supporting links, screenshots, or related issues, to provide further clarity. This structured approach facilitates efficient review and prioritization by developers. For a broader understanding of how issues and features are managed, refer to Issue and Feature Management.