YEN: A Terminal-First IDE for macOS

Maintained reference: This whitepaper is the architectural reference copy for the public YEN story. It reflects the shipped product at the v1.000 milestone and is updated as the underlying system changes.

Public PDF: https://yen.chat/whitepaper.pdf

1. Introduction

I built YEN around a simple thesis: most IDEs start from the editor and bolt on a terminal panel, but serious engineering work keeps collapsing back into the terminal anyway.

That is where builds fail. That is where services restart. That is where migrations get run, logs get tailed, branches get reviewed, containers get launched, and dangerous commands should feel dangerous before you press Enter.

So I took the opposite approach.

YEN starts with a fast native terminal on macOS and adds IDE capabilities around that terminal-first workflow. The goal is not to imitate a browser-shaped workspace. The goal is to make the terminal contextual, trustworthy, and capable enough that it can stay the center of gravity for everyday engineering work.

That leads to a few non-negotiable principles:

- The core experience should be local-first.
- Native macOS capabilities should be used directly when they offer a better result than cross-platform abstractions.
- Cloud surfaces should be optional and explicit, not a requirement for the product to function.
- The terminal should remain the primary interaction model even when richer native UI is justified.

YEN is therefore not just a terminal emulator, and it is not trying to become a browser shell for developer tooling. It is a terminal-first IDE platform with a local-first core.

2. The Thesis

The conventional IDE model looks like this:

```
Editor
  |
  +-- terminal panel
  +-- search
  +-- code intelligence
  +-- review tools
  +-- automation
```

YEN inverts it:

```
Terminal
  |
  +-- file browser
  +-- command palette
  +-- local code intelligence
  +-- project-aware search
  +-- review and merge workflows
  +-- collaboration and sharing
  +-- native macOS utility surfaces
```

This difference matters because the terminal is where code meets reality. A product built around that assumption makes different decisions about performance, trust, navigation, and workflow design.

3. Rendering and Native macOS Depth

YEN is built on a GPU-accelerated terminal engine rendered through Metal. The underlying engine is written in Zig and provides the low-latency rendering path that makes the terminal feel immediate rather than layered and heavy.

I did not write the renderer from scratch. I vendor the upstream source as an unmodified mirror and layer YEN-specific behavior on top of it. That decision keeps the rendering foundation fast-moving while letting me push the product in a very different direction.

Around that core, I use native macOS APIs directly in Swift for the surfaces that benefit from them:

- global speech-to-text hotkeys and permission orchestration
- settings UI, command palette entry points, and utility panels
- notifications, keychain access, and onboarding flows
- native helpers for video, PDF, and SVG preview paths
- screenshot, workspace, and window-management integrations

That native-first rule also changes bundle economics. Replacing heavier preview and archive dependencies with native helpers built on AVFoundation, PDFKit, AppKit, and system archive tooling saves roughly 258.8 MB compared with shipping the heavier stack directly in the app bundle.

This is an intentionally macOS-first product. The trade-off is reach. The benefit is depth.

4. Vendor-Don't-Fork Architecture

The most important architectural pattern in YEN is vendor, do not fork.

At the source layer, the upstream terminal engine lives in **ghostty/** as an exact mirror. I do not hand-edit it. All product-specific changes live in **yen-terminal/overlays/** and are applied during build time.

At the binary layer, I vendor carefully selected third-party tools as signed bundled binaries rather than re-implementing them or forcing users to install them independently. That includes the Yazi-based file browser and other bundled CLI utilities. When native macOS APIs are sufficient, I prefer small native helpers built in-repo instead of adding another dependency.

The pattern looks like this:

```
Upstream source or upstream binary
            |
            v
   YEN overlay / integration layer
            |
            v
   verification + signing + release pipeline
            |
            v
         shipped app
```

This architecture changes the maintenance economics of the project.

Instead of maintaining a permanent fork tax, I maintain explicit coupling points. If upstream changes a path or API I rely on, the build or verification step fails loudly. That is preferable to silent drift, and it is much easier to reason about than a giant long-lived fork.

The result is a project that can absorb upstream improvements quickly while still feeling opinionated and deeply customized.

5. Terminal-First IDE Surface

The product thesis becomes concrete in the IDE layer.

YEN now ships a broad terminal-first IDE surface that begins in the terminal and only escapes into native UI when that genuinely improves the job.

Project awareness

The first layer is context. **yen ide detect**, **yen ide doctor**, **yen ide trust**, **yen ide env**, **yen ide history**, **yen ide replay**, and **yen ide profile** give the terminal enough project awareness to stop feeling generic.

The terminal should know where it is, which project it belongs to, when a wrapper path should resolve to a concrete nested workspace, what environment and runtime surfaces it can trust, what readiness or security blockers exist, and what commands have already been run there.

Local code intelligence

The second layer is code intelligence that stays honest about its source.

YEN ships explicit local LSP lifecycle commands for trust, start, status, diagnostics, hover, definition, references, rename, and bounded install management. When live LSP is not available, syntax-aware AST fallback remains available for Python, JavaScript, TypeScript, Go, and Swift projects.

The important design decision is that the system does not pretend these are the same thing. Live LSP and AST fallback are labeled distinctly. Managed-vs-toolchain LSP ownership is explicit. Unsupported or unavailable states are surfaced plainly instead of being hidden behind vague “smart” language.

Search, review, and execution discipline

The command palette and project search layer treat engineering artifacts as first-class objects. Actions, files, content, diagnostics, history, review hunks, and test or verify findings all participate in one shared workflow surface.

That surface now speaks one reusable semantic location language: **path:line[:col]**. Search results, grouped diagnostics, test failures, verify evidence, quality output, and review hunks can all point at the same kind of target and be reused across CLI and native review flows.

That same philosophy extends to verification, approval gating, repo-native workflow discovery with explicit task execution, grouped test status and reruns, format / lint / fix execution, timeline auditing, PR review, and merge-conflict handling. When YEN falls back, crosses a trust boundary, or needs repository-implied companion commands for polyglot execution, it says so explicitly instead of pretending the workflow was simpler than it was.

This is one of the core novelties of YEN: the IDE does not live in a separate editor shell. It is progressively layered into terminal-native flow.

Collaboration and migration

YEN also includes project-scoped sharing and collaboration primitives, local devcontainer control, deterministic VS Code workspace inventory / import paths, and native macOS PR review and merge-conflict workspaces that stay anchored to the same CLI trust and evidence model. These are trust-gated and intentionally explicit.

The philosophy is the same throughout the stack: make the workflow available, but do not smuggle in hidden authority.

6. On-Device Speech-to-Text

Speech-to-text is one of the clearest expressions of the YEN philosophy.

The feature is global. Press **Option + Space** in any macOS app and YEN can capture audio, transcribe it on-device through Apple's speech stack, and insert the result back into the active context.

On newer macOS releases, YEN uses Apple's more modern speech path with better behavior and fewer session limits. On older supported macOS versions, it falls back to the legacy recognizer path and handles those constraints explicitly.

The feature is useful because it is built for terminal and system reality, not a marketing bullet:

- it is on-device rather than routed through a third-party speech API
- it has `Live Transcript Preview` and recovery states
- it can stay available via launch-at-login behavior without an open terminal window
- it treats permission state as part of the product, not as an afterthought
- it can optionally translate dictated text before insertion

The implementation detail that matters most is not the exact framework stack. It is the product stance: native capability, local processing, explicit diagnostics, and no guesswork when something is blocked.

7. Productive on First Launch

YEN is designed to feel like an environment, not a blank terminal shell.

That means the app ships with a large amount of useful surface area already integrated:

Area	What ships
File workflows	Built-in file browser, previews, cd-on-quit behavior
Navigation	Command palette, tab sidebar, scratchpad, split layouts, split labels
Customization	Curated themes, live settings, notification sound system
Bundled tools	Search, fuzzy matching, git / system tooling, native helpers
Communication	Chat, mail, and calendar TUIs
IDE workflows	doctor / trust / verify, LSP lifecycle, AST fallback, workflow and task execution, test + quality baselines, sharing, devcontainer control, VS Code import, review tools

The point is not feature accumulation for its own sake. The point is to reduce setup friction while keeping the terminal clean. Features should be present when needed and mostly invisible when not.

The file browser is a good example of that philosophy. YEN keeps the integration thin and shell-friendly, uses vendored Yazi binaries for the terminal-native browser path, and relies on native preview helpers and cd-on-quit behavior instead of turning file navigation into a detached GUI-only subsystem.

8. Communication Surfaces

YEN ships three communication-oriented terminal clients as separate Go TUIs:

- chat
- mail
- calendar

Each exists because it benefits from a terminal-native interaction model and because it reinforces the larger idea that the terminal can host more of a developer's working environment than people usually assume.

The architecture is intentionally specific:

- Chat talks through YEN-owned HTTP API surfaces, with anonymous direct database access intentionally denied.
- Mail and Calendar use OAuth with PKCE and store tokens in the macOS keychain.
- Notification delivery is service-specific rather than forced through one fake-unified bridge.

The result is a cleaner trust model. Remote services are used where they are inherently required, but the product does not pretend that "local-first" means "never uses a network." It means the local product remains primary and explicit about what crosses the network and why.

9. Privacy and Trust Model

YEN's privacy model is not "everything is local." That would be false.

The correct model is:

- the core product is local-first
- cloud dependency is not required for the terminal itself
- remote services are opt-in and feature-specific
- trust boundaries are explicit

The terminal, file browser, settings, themes, layout tools, screenshot tooling, and large parts of the IDE layer work locally. Features that require remote systems, such as Gmail, Google Calendar, chat backends, or external collaboration adapters, are separate by design.

This matters because modern developer tools often hide their trust model behind convenience language. YEN tries to do the opposite. Project trust is explicit. LSP trust is explicit. Sharing trust is explicit. Workflow trust is explicit. Doctor, trust, and verify status all reuse the same local sources of truth instead of inventing parallel reassurance layers. OAuth-backed services are explicit.

That honesty is part of the product.

10. Build and Release Automation

The release system is another core part of the architecture.

YEN ships through a 12-step release workflow covering preflight, build, verification, packaging, notarization, upload, staging, commit discipline, and final push. The goal is not just automation. The goal is repeatable automation with explicit gates.

That workflow verifies things that are easy to get wrong in macOS desktop software:

- vendored dependency freshness
- component integrity before build
- overlay application
- version propagation
- signing order
- notarization and stapling
- appcast generation
- release artifact integrity
- staged file discipline

The release system is also where the vendor-don't-fork philosophy pays off. Because the project spans Zig, Swift, Go, Shell, SQL, and web code, the release harness has to be more disciplined than a typical single-language app build.

The right summary is simple: shipping quickly only works when verification is better than bravado.

11. Language Selection

YEN uses multiple languages because the project has multiple kinds of constraints.

Component	Language	Why it belongs there
Terminal engine	Zig	low-level performance and rendering control
Native app surfaces	Swift	direct access to macOS frameworks
Communication TUIs	Go	strong terminal-app ergonomics and simple binaries
CLI and automation	Shell	immediate startup and zero-runtime dependency distribution
Web and docs surface	TypeScript	strong web tooling and static / SSR delivery
Database evolution	SQL	explicit schema ownership

This is not a single-language ideology project. It is a pragmatic system where each layer uses the tool that best matches the job.

The trade-off is build complexity. The benefit is that each subsystem can be built around the constraints that actually matter to it.

12. Tradeoffs

YEN makes several tradeoffs on purpose.

macOS focus. The product goes deep on Apple APIs instead of chasing immediate cross-platform breadth.

Explicit trust instead of silent convenience. Some workflows require a little more ceremony because I want the user to know what is being trusted and why.

Overlay maintenance instead of fork sprawl. The vendor-don't-fork pattern still has coupling points. They are just easier to manage than a giant diverging fork.

Terminal-first constraints. Some tasks really do benefit from richer native UI, so YEN uses them selectively. The product is not dogmatic about "everything must be a terminal pane."

Feature breadth with verification burden. Every additional shipped surface increases the testing and release burden. That is the real cost of building a product like this as a solo developer.

These are acceptable tradeoffs because they reinforce the central thesis rather than diluting it.

13. Conclusion

YEN is the result of treating the terminal as a primary software surface rather than a utility panel.

That changes architecture. It changes trust boundaries. It changes the role of native UI. It changes what "productive on first launch" means. It changes how IDE workflows are shaped. And it changes how much of the developer environment can live inside one coherent product without becoming a cloudy, account-gated control plane.

I built YEN to prove that the terminal can be more than a fast text box and that an IDE can begin with the shell instead of apologizing for it.

The recent IDE hardening work mattered because it turned a broad feature list into a more coherent system: one trust story, one evidence model, and one semantic location language that can move between CLI and native surfaces without changing shape.

That is the product thesis.

At v1.000, it is no longer a theory.

References

1. **README.md** — product overview and architectural thesis
2. **docs/02-architecture.md** — vendor pattern and system structure
3. **docs/03-build-desktop.md** — build, signing, notarization, and release workflow
4. **docs/05-commands.md** — commands, shortcuts, CLI action surface, and IDE helpers
5. **docs/06-chat.md** — chat architecture and security model
6. **docs/07-mail-cal.md** — mail / calendar architecture and OAuth model
7. **docs/09-qol.md** — shipped feature surface
8. **docs/12-ide.md** — terminal-first IDE thesis and current research / roadmap
9. **docs/10-vendor-audit.md** — bundled binary posture, overlap audit, and native-first size economics
10. **docs/14-file-browser.md** — file-browser architecture, plugin guardrails, and launch model
11. Apple Developer Documentation — Speech, AppKit, AVFoundation, PDFKit, Metal