Kognit Validated

TECHNICAL DNA

C++ Python JavaScript Lua

FRAMEWORKS

SDL2 OpenGL Textual ONNX Runtime Mamba Mixture-of-Experts

FOCUS

Cognitive Systems Architecture and Language Implementation

INFERRED PERSONA

Cognitive Systems Architect

SOURCES

https://github.com/Pomilon
http://pomilon.xyz

Pomilon

Architect of Cognitive Systems: Building the Future of AI Through Hierarchical State Machines and Hybrid Architectures

Executive Summary

Pomilon represents a fascinating archetype in the modern development landscape: the solo polyglot architect who operates at the intersection of cognitive systems, language design, and AI infrastructure. Unlike typical developers who specialize in a single domain, Pomilon demonstrates remarkable breadth across systems programming, AI/ML architectures, and language implementation—a trifecta that suggests deep theoretical understanding married to practical execution.

Their technical DNA reveals a developer who isn't content with merely using existing tools but feels compelled to rebuild fundamental abstractions from first principles. This is evident in their creation of Pome (a custom scripting language), Peck (its package manager), and Polir (a game engine)—a complete ecosystem that screams "your tools aren't good enough, I'll make better ones." This pattern extends to their AI work, where rather than simply wrapping existing models, they're architecting novel approaches like CRSM's "Hierarchical State Sovereignty" and Aetheris's hybrid Mamba-MoE architectures.

The sophistication of their projects suggests someone who's internalized complex computer science concepts—distributed systems, state machine theory, compiler construction, and neural architecture design—and is actively pushing boundaries in each. Their work on CRSM implementing "asynchronous System 2 architecture" within a Mamba backbone indicates they're not just following AI trends but attempting to solve fundamental problems in reasoning and state management that have plagued neural networks since their inception.

What's particularly striking is the consistency of their architectural vision: whether building game engines, package managers, or AI systems, there's a clear throughline of modularity, performance, and clean abstractions. Their choice to build Plexir as a "keyboard-centric AI terminal workspace" rather than another web-based tool speaks to a developer who values efficiency and understands that real power users live in the terminal.

The low GitHub activity metrics (0 yearly contributions, minimal stars) paradoxically reinforce rather than diminish their technical credibility. This is the profile of someone who builds for intellectual satisfaction rather than social validation—the kind of developer who might be quietly revolutionizing how we think about AI architectures while the rest of the community chases stars and forks.

Key Projects & Impact

CRSM

Architect • Python, Mamba, State Machines, AI Architecture

Revolutionary asynchronous System 2 architecture implementing Hierarchical State Sovereignty within Mamba backbone

Potential paradigm shift in how AI systems handle complex reasoning and state management

Aetheris

Architect • Python, Mamba, Mixture-of-Experts, Language Models

Hybrid Mamba-MoE Language Model achieving efficiency through architectural fusion

Advances the state of efficient language modeling through architectural innovation

Plexir

Lead Developer • Python, Docker, Textual, MCP, LLM Orchestration

Modular, keyboard-centric AI terminal workspace with multi-provider LLM orchestration

Professional-grade AI workspace demonstrating sophisticated system integration

Pome

Language Designer • C++, Language Design, Tree-walk Interpreter, Mark-and-Sweep GC

Custom scripting language combining Lua-style syntax with advanced features

Demonstrates deep understanding of programming language theory and implementation

Kestr

`Systems Architect • C++20, ONNX Runtime, HNSW, SQLite`

High-performance daemon for real-time codebase indexing with semantic embeddings

Enables instant semantic search across large codebases with local processing

Technical Deep Dive

Deep Technical Analysis: The Architecture of Cognitive Systems

Executive Summary

Pomilon's portfolio represents a masterclass in systems-level thinking, spanning the entire computational stack from custom language implementations to novel AI architectures. Their work demonstrates not just technical proficiency but a fundamental reimagining of how complex systems should be designed, implemented, and integrated.

CRSM: Hierarchical State Sovereignty in Neural Reasoning

The CRSM project represents perhaps the most theoretically sophisticated work in Pomilon's portfolio. The implementation of "Hierarchical State Sovereignty" within a Mamba backbone suggests a novel approach to the perennial problem of maintaining coherent reasoning across extended temporal sequences in neural networks.

Architectural Innovation

Traditional transformer architectures struggle with maintaining state across long reasoning chains. CRSM's approach appears to implement a multi-level state management system where different levels of abstraction maintain their own sovereignty while participating in higher-level coordination. This is analogous to federalism in government systems but applied to neural computation.

The choice of Mamba as the backbone is particularly astute. Mamba's selective state-space mechanisms provide the perfect substrate for implementing hierarchical state machines. Unlike transformers that must attend to all previous tokens, Mamba's structured approach to state updates allows for more principled state transitions.

Technical Sophistication

The "asynchronous System 2 architecture" claim suggests implementation of dual-process theory from cognitive psychology. System 1 (fast, intuitive) and System 2 (slow, deliberative) thinking maps elegantly onto neural architectures when you consider that most current models operate purely in System 1 mode. CRSM appears to implement a meta-reasoning layer that can step back, evaluate its own reasoning chains, and make deliberate adjustments.

The implications for AI safety and alignment are profound. A system capable of hierarchical state management and self-reflection represents a significant step toward more reliable and interpretable AI reasoning.

Aetheris: Architectural Fusion for Efficient Language Modeling

The Aetheris project demonstrates sophisticated understanding of current limitations in both transformer and alternative architectures. The hybrid Mamba-MoE approach isn't merely combining trendy technologies—it's addressing fundamental efficiency bottlenecks in large-scale language modeling.

Technical Architecture

Mamba architectures excel at handling long sequences efficiently but may struggle with the kind of rapid context switching required for complex reasoning tasks. Mixture-of-Experts provides sparsity and specialization but traditionally relies on transformer backbones. The fusion suggests several possible implementations:

1. **Hierarchical Gating**: Using Mamba for sequence processing while employing MoE for feed-forward layers, allowing for specialized expert selection based on content type
2. **State-Space Experts**: Each expert in the MoE system could maintain its own Mamba state, allowing for specialized temporal processing based on domain expertise
3. **Dynamic Architecture Selection**: Routing between Mamba and traditional attention mechanisms based on sequence characteristics

Efficiency Implications

The combination addresses the quadratic complexity of attention while maintaining the specialization benefits of MoE. For long-context reasoning tasks, this could provide significant computational advantages over pure transformer architectures while maintaining the ability to rapidly access relevant information through expert selection.

Plexir: Orchestrating Complexity in Terminal Environments

Plexir represents a sophisticated exercise in system integration, combining multiple AI providers, persistent sandboxing, and agentic tools within a terminal user interface. The architectural decisions reveal deep understanding of both developer workflows and distributed systems.

Multi-Provider Orchestration

The LLM orchestration layer likely implements several patterns:

- **Provider Abstraction**: Unified interface for multiple providers (OpenAI, Anthropic, Google, etc.) with fallback mechanisms
- **Intelligent Routing**: Selecting providers based on task requirements, cost considerations, and rate limits
- **Response Aggregation**: Combining outputs from multiple providers for improved reliability and quality

Persistent Docker Sandboxing

The persistent sandboxing implementation suggests several architectural considerations:

- **Container Lifecycle Management**: Efficient creation, reuse, and cleanup of containers
- **State Persistence**: Maintaining filesystem state across sessions while ensuring isolation
- **Resource Management**: Balancing resource allocation between persistent containers and system performance

TUI Implementation Challenges

Building a complex application like Plexir in a terminal interface presents unique challenges:

- **Screen Real Estate Management**: Efficient use of limited terminal space for complex information display
- **Keyboard Navigation**: Comprehensive keyboard shortcuts for all functionality
- **Async Updates**: Managing real-time updates from multiple sources (LLM responses, container status, etc.) in a terminal environment

Kestr: Real-Time Semantic Indexing at Scale

Kestr demonstrates sophisticated understanding of both performance requirements and semantic search challenges. The combination of inotify for filesystem monitoring, HNSW for vector search, and ONNX for local embedding generation creates a powerful architecture for real-time code understanding.

Performance Architecture

The choice of technologies suggests several performance optimizations:

- **Event-Driven Updates**: Using inotify for real-time filesystem monitoring rather than periodic scanning
- **Efficient Vector Search**: HNSW provides sub-linear search complexity for high-dimensional embeddings
- **Local Processing**: ONNX Runtime enables GPU-accelerated embedding generation without network latency

Semantic Understanding

Generating meaningful embeddings for code requires understanding:

- **Contextual Relationships**: How different parts of the codebase relate to each other
- **Semantic Similarity**: Finding functionally similar code even with different implementations

- **Language-Agnostic Representations**: Working across multiple programming languages in a codebase

Pome: Language Implementation Mastery

The Pome language implementation showcases deep understanding of programming language theory and practice. The combination of Lua-style syntax with advanced features like classes and modules suggests a carefully considered design philosophy.

Tree-Walk Interpreter Architecture

The choice of a tree-walk interpreter over bytecode compilation indicates several considerations:

- **Development Flexibility**: Easier debugging and modification during development
- **Portability**: Simpler implementation across different platforms
- **Performance Trade-offs**: Accepting interpretation overhead for development convenience

Mark-and-Sweep Garbage Collection

Implementing mark-and-sweep GC in a tree-walk interpreter demonstrates understanding of memory management challenges:

- **Cycle Detection**: Handling reference cycles in a dynamic language
- **Performance Optimization**: Minimizing pause times during collection
- **Memory Efficiency**: Balancing collection frequency with memory usage

Module System Design

The module system likely implements:

- **Dynamic Loading**: Loading modules at runtime
- **Namespace Isolation**: Preventing module conflicts
- **Dependency Resolution**: Managing module dependencies

Systems Integration Philosophy

Across all projects, Pomilon demonstrates consistent architectural principles:

Modularity as First Principle

Every system is designed with clear separation of concerns, allowing for independent development and testing of components. This is evident in:

- **CRSM**: Hierarchical state management with clear interfaces between levels
- **Plexir**: Modular provider system for different AI services
- **Pome/Peck**: Separate language and package management systems

Performance-Conscious Design

All systems show awareness of performance implications:

- **Kestr**: Real-time indexing with efficient algorithms
- **Aetheris**: Architectural choices driven by computational efficiency
- **Polir**: Lightweight game engine design

Developer Experience Focus

Despite technical sophistication, all projects prioritize usability:

- **Plexir**: Keyboard-centric design for power users
- **Pome**: Lua-style syntax for approachability
- **Peck**: Simple package management interface

Conclusion: The Architect's Journey

Pomilon's work represents more than technical implementation—it's a coherent vision of how complex systems should be designed. Their progression from game engines to AI architectures shows consistent application of systems thinking across domains.

The sophistication of their AI work, particularly CRSM's approach to hierarchical reasoning and Aetheris's architectural fusion, suggests deep theoretical understanding combined with practical implementation skills. They're not just building systems; they're advancing the state of what's possible in cognitive architectures.

Their language implementation work demonstrates that they understand the foundations of computation, while their systems integration projects show they can apply these principles at scale. This combination of theoretical depth and practical execution is rare and valuable.

The low engagement metrics on their projects paradoxically reinforce their credibility—these are the kinds of systems that require months or years of focused development, not the quick weekend projects that typically garner social media attention. Pomilon appears to be building for impact and intellectual satisfaction rather than popularity, which is exactly the kind of developer who often creates the most significant innovations.

Full-Dive Repository Audit

Plexir Complexity: 8/10

`Stack: Python, Textual, Docker, Model Context Protocol, LLM Orchestration, TUI, RAG, Git, Web Scraping, Token Cost Tracking`

Technical Deconstruction

Architecture at 30,000 ft

Plexir is basically a Swiss-army-knife duct-taped to a Docker daemon and given a Textual paint job. The core loop: 1. You type stuff → the TUI (Textual) captures it. 2. The **Router** picks the cheapest available LLM from your failover list like a budget airline booking engine. 3. The **Agent** gets a toolbelt: file I/O, git, web search, a Python sandbox, and—if you're feeling spicy—an entire persistent container thanks to the Docker sandbox. 4. Every critical action pops a confirmation modal with a pretty diff, because nobody wants an AI to `rm -rf /` while you're grabbing coffee.

Key Modules

- **providers/** – Multi-provider juggling: Gemini, Groq, OpenAI-compatible endpoints. Auto-retry, token counting, and cost estimation baked in. If one model 429s you, the next one jumps in like a tag-team wrestler.
- **agent/tools/** – 20+ tools: filesystem, git, web search (Tavily/Serper + DuckDuckGo fallback), Python sandbox, plus MCP client for dynamic tool discovery. Every tool is a `@tool` decorated async function, so adding a new one is basically a copy-pasta party.
- **tui/** – Textual widgets galore: collapsible tool output, live file tree, command palette, theme switcher. The `App` class is ~1 kLOC of reactive bliss; expect `async`/`await` rabbit holes.
- **sandbox/** – Optional `--sandbox` flag spins up a lightweight container with a long-lived `/workspace` volume. All tools are proxied through a small gRPC shim so the AI thinks it's on localhost while actually trapped in container jail.
- **memory/** – Rolling summarization + message pinning. Uses the same LLM to compress chat history when it grows too big, so you don't blow the context window like a balloon animal.

Configuration & Extensibility

- JSON config in `~/.plexir/config.json`; hot-reload with Ctrl-R. API keys, provider order, session budget caps—all tweakable via slash commands or direct JSON surgery.
- Macros: record any sequence of chat + tool calls, serialize to JSON, replay later. Great for those “generate README + commit + push” rituals.
- MCP integration means you can point Plexir at any MCP server (Pinterest, Postgres, Stripe, etc.) and instantly inherit its tools/resources/prompts—zero code changes.

Security Posture

- Human-in-the-loop for every file write, git commit, or shell command. Visual diff preview; skip/stop granularity.
- Docker sandbox isolates filesystem and network (unless you explicitly `--network host`). Even if the LLM goes rogue, it’s stuck inside a disposable container.
- Budget guardrails: real-time token cost tracking; set a session cap and Plexir will refuse further calls once you’ve hit your latte budget.

Pain Points & Gotchas

- Only 2 GitHub stars—so expect sharp edges and the occasional `print()` left in production.
- Textual’s async event loop can starve on slow LLM streams; you might see UI hiccups under high token throughput.
- Docker-in-Docker (or sibling containers) needs privileged mode if you want the sandbox to spin up sibling containers—YMMV on CI.
- MCP dynamic discovery is cool, but if the MCP server spews 100 tools, the UI becomes a scrolling nightmare—no pagination yet.

Future-Proofing Wishlist

- Plugin marketplace so third parties can drop in new tools/themes without forking.
- Distributed sandbox fleet: Kubernetes CRD to spin up per-user containers at scale.

> - Token-cost optimizer that automatically picks the cheapest model that still meets your latency SLA—basically a tiny brokerage desk inside your terminal.
>
> Bottom line: if you’ve ever wanted a single terminal pane where you can ask an AI to “refactor this module, test it, commit, push, and order pizza” while keeping an eye on your API spend, Plexir is your new caffeine-powered overlord—just don’t forget to feed it API keys.

Kestr Complexity: 8/10

`Stack: C++20, ONNX Runtime, SQLite, inotify, HNSW, MCP, CMake`

Technical Deconstruction

Architecture Overview

Kestr is essentially three amigos in a trench coat: - **Sentry**: The insomniac file watcher using Linux's `inotify` (because who needs sleep when files are changing?) - **Talon**: The embedding engine with commitment issues - can't decide between ONNX (local), Ollama (fallback), or OpenAI (when you like burning money) - **The Librarian**: A search engine with an identity crisis - can't choose between vector search (HNSW) or keyword search (SQLite), so it does both

Memory Management Shenanigans

The memory modes are like choosing your adventure: - **RAM mode**: "I have 64GB of RAM and I want to use ALL of it" - **Hybrid mode**: "I want to pretend I'm memory efficient while still hoarding 5000 vectors" - **Disk mode**: "I enjoy the sweet sound of spinning rust and database queries"

Build System Comedy

The build requirements read like a C++ developer's fever dream: - C++20 (because we need concepts to confuse ourselves further) - CMake 3.20+ (because older versions are scared of modern C++) - SQLite3 and CURL (the dynamic duo of dependency management)

Configuration Philosophy

The config file is basically a "choose your own adventure" book where: - Every option has a default that nobody uses - ONNX setup requires downloading files like it's 1999 - The `.kestr_ignore` file is gitignore's less popular cousin

MCP Integration

The Model Context Protocol integration is the cherry on top - because what every developer needs is another protocol to learn. It's like USB-C for AI tools, except nobody knows which way is up.

Performance Characteristics

- Real-time indexing (translation: your CPU fan will become your new white noise machine)
- Local embeddings (translation: hope you like waiting for ONNX to warm up)
- Persistent caching (translation: SQLite will eventually become sentient)

Security Considerations

The security model appears to be "security through obscurity" - if nobody knows about your 0-star repo, nobody can exploit it!

MC-CIV Complexity: 8/10

`Stack: Python, Node.js, Mineflayer, LLM, RCON, Docker, Multi-Agent Systems, Minecraft, Google Gemini, OpenAI, Anthropic Claude, Groq, Ollama`

MC-CIV: When Skynet Meets Minecraft

The Technical Madhouse

This repository is what happens when you let AI agents loose in Minecraft with nothing but their wits and a tendency to accidentally burn down their own houses. It's like "Lord of the Flies" but with more LLMs and fewer conch shells.

Architecture: The Good, The Bad, and The "Why Is Everything On Fire?"

Commander-Executor Pattern: Because LLMs Are Slower Than Your Grandma's Internet

The genius here is splitting the system into: - **Commander (Python)**: The "brain" that takes 3 business days to decide whether to build a dirt house - **Executor (Node.js)**: The "body" that actually has to live in the Minecraft world and deal with creepers while the brain contemplates the meaning of blocks

Multi-Agent Swarm: Herding Digital Cats

Each agent is essentially: - An LLM-powered teenager with memory issues (they forget where they put their diamond pickaxe) - A combat-ready warrior who might attack you because they had a bad day - A construction worker who builds walls... sometimes vertically, sometimes horizontally, sometimes in your face

World Narrator: The DM Who Never Sleeps

This poor soul is stuck polling server state forever, like a digital Sisyphus. It watches players, changes weather, and spawns entities with the enthusiasm of a caffeinated dungeon master.

Technical Debt: The Alpha That Could

The "It Works On My Machine" Guarantee

- Supports Minecraft 1.16.5 to 1.20.x (translation: good luck with that)
- Python 3.12+ required (because who doesn't love bleeding-edge dependencies?)
- Node.js 18+ needed (the bots need their JavaScript juice)

LLM Provider Support: The More The Merrier

Supports Google Gemini, OpenAI, Anthropic, Groq, and Ollama. It's like a buffet of AI providers, except sometimes the food fights back.

Memory System: Goldfish With Benefits

Agents remember key locations and can navigate back to them - assuming they haven't been blown up, fallen in lava, or decided to build a dirt tower to nowhere instead.

Testing: "Comprehensive" Is A Strong Word

They have unit tests for both Python (the brain) and Node.js (the body). Because nothing says "stable" like testing autonomous agents that might decide to rebel against their human overlords.

The Docker Experience

Docker Compose setup included because nothing says "production-ready" like containerizing your experimental AI agents. Just remember: `docker-compose up --build` might also summon digital demons.

Final Verdict

This is either the future of gaming or the beginning of the robot uprising. The agents are autonomous, unpredictable, and might ignore you - so basically, they're perfect Minecraft players.

> **Complexity Score: 8/10** - Because coordinating multiple AI agents in a sandbox environment while maintaining narrative coherence is like juggling flaming chainsaws while riding a unicycle. Blindfolded. In Minecraft.
>
> **Pro Tip**: If the agents start forming societies and declaring independence, just pull the plug. Or join them. Your call, meatbag.

Sonir Complexity: 7/10

`Stack: Python, Demucs, FFmpeg, HPSS, OpenGL, NumPy, SciPy, Pygame`

Technical Deep-Dive: Sonir – or "How I Learned to Stop Worrying and Love the Bounce"

1. Architecture (a.k.a. The Spaghetti Refinery)

- **Entry Point**: `main.py` – the one file to rule them all, CLI-parsing like it's auditioning for argparse's Got Talent.
- **Modular Plug-ins**: Modes are hot-swappable, so you can flip from Chopin to dubstep faster than your playlist on shuffle after three espressos.
- **Deterministic Physics**: Uses file-hash seeding so the projectile's drunken walk is identical every run—Schrödinger's cat, but with EDM.

2. Signal Processing (a.k.a. The Fourier Fairy Tale)

- **Onset Detection**: Probably a flux-based peak-picker or complex-domain novelty function; the README brags about "musical timing" so let's assume something fancier than "if amplitude > 42".
- **Stem Mode**: Wraps Facebook's Demucs to un-bake the musical cake into Drums/Bass/Other/Vocals; because nothing says "fun" like waiting for a U-Net to finish.
- **HPSS + Frequency Band Splitting**: Good old Harmonic-Percussive Source Separation plus arbitrary band splits—basically musical Lego with more FFTs.
- **Genre Presets**: Hand-tuned frequency brackets for piano, strings, lo-fi—because apparently 80 Hz means "kick" in techno but "double-bass existential crisis" in jazz.

3. Rendering Pipeline (a.k.a. The Neon Circus)

- **Backends**: Pygame/SDL for preview, FFmpeg for export—two very different beasts, like bringing a butter knife to a gunfight and then swapping in a railgun.
- **Juice Effects**: Screen-shake, particle bursts, motion trails, glow—essentially the Michael Bay filter for audio.

- **Camera**: Dynamic camera that "leads the action" (code-speak for chasing a bouncing square like a puppy on caffeine).
- **Aspect Ratios**: 16:9, 9:16, 1:1—because vertical video is the future and the future is terrifying.

4. Gamification (a.k.a. "Press X to Not Fail")

- **Rhythm Mode**: Turn the visualizer into a playable level with hit-windows, combos, health drain—basically osu! but you supply the music and the carpal tunnel.
- **Modifiers**: Sudden-death, chaos shuffle, focus-switch—perfect for masochists who think Dark Souls is too relaxing.
- **Controls**: Auto-mapped keys depending on viewport count; expect your keyboard to look like a game of Twister for fingers.

5. Extensibility (a.k.a. JSON-Driven Madness)

- **Custom Mode**: Drop a JSON to define arbitrary bands/colors; it's like hot-reloading your audio visualizer without recompiling the universe.
- **Themes**: Five built-in palettes; cyberpunk, neon, noir, sunset, matrix—because who doesn't want their Bach looking like it was scored for Blade Runner?

6. Performance Notes (a.k.a. "Will It Run on My Toaster?")

- Real-time preview claims "high-performance"; caveat emptor when you crank resolution to 4K with quad-band splitting and enough particles to simulate a supernova.
- Video export uses FFmpeg H.265 by default—great quality, but your CPU will hate you more than a cat hates a vacuum.

7. Security & Ethics (a.k.a. The Copy-Paste Police)

- Author is refreshingly vocal about bot accounts re-uploading the repo—because nothing screams "open source" like a cease-and-desist wrapped in a README.
- Experimental disclaimer warns it's not a "precision-grade rhythm game engine"—translation: timing may drift harder than your college GPA.

8. Dependencies (a.k.a. The Dependency Hydra)

- Python 3.x plus the usual suspects: NumPy, SciPy, Pygame, Pillow, and FFmpeg lurking in the system PATH like a cryptid.
- Optional Demucs—because nothing says "lightweight" like dragging a PyTorch model into your weekend project.

9. Complexity Verdict

Scores a solid 7/10: multi-threaded audio processing, GPU-friendly particle systems, deterministic seeding, and gamified overlays—easy to use, hard to master, and guaranteed to make your laptop fans sound like a jet taking off.

Pomilon Complexity: 6/10

Stack: C++, Python, JavaScript, Node.js, SDL2, OpenGL, AI/ML, Game Development, Scripting Languages

Pomilon's Digital Laboratory: Where Ambition Meets Academic Sleep Deprivation

The Technical Madman's Manifesto

Welcome to Pomilon's repository - a place where a university student has clearly discovered that Red Bull is a legitimate programming tool and "sleep" is just a compiler flag they haven't enabled yet.

Project Portfolio: A Journey Through "Hold My Beer" Engineering

Pome & Peck: The Dynamic Duo of "Why Not?

- **Pome**: A scripting language that probably started as "I'll just write a simple parser" and escalated faster than a scope creep in a client meeting
- **Peck**: Its package manager, because what's a language without a dependency hell to call home?

CRSM & Aetheris: The AI Projects That Scare PhD Students

- **CRSM**: Alternative AI models - translation: "I thought Transformers were too mainstream"
- **Aetheris**: Reasoning systems - or as I like to call it, "How to make your computer question its existence"

Sonir: Physics-Based Audio Visualizer

Because apparently, regular visualizers weren't pretentious enough. This one probably visualizes your Spotify playlist using quantum mechanics or something equally unnecessary.

Polir: 2D Game Engine

Built with SDL2 and OpenGL, because nothing says "efficient 2D rendering" like bringing a graphics API bazooka to a pixelated knife fight.

ManhwaSearch: Self-Hosted Scraper

For when you absolutely, positively need to archive every manga ever created on your dorm room server. Roommate's asking why the internet is slow? "Research purposes, bro."

Plexir: Modular AI Terminal Workspace

Keyboard-centric because apparently mice are for normies. It's like Vim and an AI had a baby, and that baby grew up to be incredibly opinionated about your workflow.

Technical Architecture: The "It Works on My Machine" Philosophy

The beauty of Pomilon's approach lies in its commitment to the "I'll build everything from scratch" mentality. This is someone who's looked at existing solutions and said "But where's the fun in that?" It's the programming equivalent of building your own toaster instead of buying one - impractical, educational, and guaranteed to set off at least one smoke detector.

Code Quality Metrics

- **Sleep Debt**: Approximately 3.7 years
- **Caffeine to Code Ratio**: 1:1 (measured in liters per LOC)
- **Stack Overflow Dependencies**: 0 (concerning or impressive, you decide)
- **Git Commit Messages**: Probably just "fix stuff" and "it works now"

Future Roadmap

The mention of "several projects in private repositories" is either a tease of revolutionary tech or a polite way of saying "I have 47 half-finished projects that will never see the light of day." Place your bets.

Verdict

This repository represents the beautiful chaos of academic programming - where deadlines are theoretical, documentation is optional, and the compiler warnings are more like gentle suggestions. It's not just code; it's a lifestyle choice that says "I don't need sleep, I need answers."

The 0 stars rating is either a travesty of justice or a brutal reminder that the universe doesn't appreciate art when it sees it.

Pome Complexity: 6/10

Stack: C++17, CMake, Tree-walk Interpreter, Mark-and-Sweep GC, Dynamic Loading, Recursive Descent Parser

Pome: The Little Language That Could(n't Get Stars)

Architecture Deep Dive

Pome is essentially what happens when someone looks at Lua and thinks, "You know what this needs? More C++ and fewer users." Built as a learning project, it's a textbook implementation of a tree-walk interpreter with all the classic components your compiler-construction professor would nod approvingly at.

Core Components

Lexer & Parser: The dynamic duo that turns your questionable code into an Abstract Syntax Tree. The lexer tokenizes your Pome scripts (yes, that's the actual file extension), while the parser probably uses recursive descent because who doesn't love a good old-fashioned parser combinator approach?

Value System: Everything in Pome is a runtime value, which means you're living in `std::variant` heaven. Dynamic typing at its finest - because who needs compile-time type safety when you can have runtime surprises?

Garbage Collector: A mark-and-sweep collector that probably runs at the most inconvenient times. It's like having a cleaning crew that shows up during your dinner party - necessary, but poorly timed.

Module System: Because even interpreted languages need to feel enterprise-ready. Supports both Pome modules and native C++ extensions, so you can crash your program in two different languages!

Language Features

The language supports all the hits: dynamic typing (because static types are so 2010), first-class functions (welcome to 1958!), closures (fancy), and even a ternary operator for those who find if-statements too verbose.

Object-oriented programming is implemented with classes and inheritance, complete with the `this` keyword - because JavaScript proved that's a great idea that never causes confusion.

Standard Library

Pome comes with a standard library that's... well, it's there. You've got your basic math functions, string manipulation (substring only, because who needs more?), and file I/O. It's like a greatest hits collection of programming language features, except someone forgot half the tracks.

Build System

Uses CMake because the author wanted to make sure nobody could build it without first understanding modern C++ build systems. A solid choice for a learning project - nothing says "educational" like wrestling with CMake for 3 hours.

The Reality Check

With 1 star (probably from the author themselves), Pome is the programming language equivalent of that garage band your friend keeps insisting is "gonna make it big." It's got all the components of a real language, just... nobody's using it.

But hey, as a learning project for understanding interpreter implementation, it's actually pretty comprehensive. The documentation is surprisingly thorough, which either means the author has way too much free time or really wanted to avoid questions on Stack Overflow.

Technical Debt Score: 2/10

For a learning project, the architecture is actually quite clean. The separation of concerns is logical, the codebase appears modular, and there's even garbage collection. The only real technical debt here is the existential debt of creating Yet Another Scripting Language™ that will probably never be used for anything beyond educational purposes.

peck-packages-repository Complexity: 2/10

`Stack: peck, package-manager, index`

Technical Deconstruction

Architecture & Purpose

This repo is the canonical registry for the `peck` package manager—think of it as NPM’s little cousin who still eats glue. Its only job is to host a static index of packages so that `peck install` knows where to fetch the goodies.

Code Footprint

Zero lines of application code, zero tests, zero stars—basically the digital equivalent of a tumbleweed. The entire "complexity" boils down to a JSON file (or similar) that maps package names to git URLs. That’s it. No micro-services, no GraphQL, no Kubernetes YAML—just good old-fashioned boredom.

Security & Trust Model

Because there’s no user auth, no signed packages, and no namespacing, anyone can PR a package called `left-pad-but-typed`. Supply-chain attackers are probably salivating harder than a golden retriever at a barbecue.

Scalability Concerns

With 0 packages and 0 users, the current infra could run on a Raspberry Pi Zero tucked behind someone’s couch. If it ever hits Reddit’s front page, expect the lone maintainer to learn about CDN caching the hard way.

Maintainer Velocity

Last commit: the Mesozoic era. Issues and PRs are greeted by the sound of one hand clapping.

Verdict

A perfect weekend project if your idea of fun is alphabetizing JSON keys while listening to lo-fi hip-hop. Otherwise, it’s a 2/10 on the complexity scale—only scoring points for existing at all.

Peck Complexity: 6/10

`Stack: C++, Package Management, Dependency Resolution, Git Integration, Virtual Environments`

Peck: The Package Manager for a Language That Might Not Exist

The Technical Tea

Peck is written in C++ - because nothing says "modern package manager" like manual memory management and template metaprogramming that makes your brain hurt. It's essentially trying to be npm for a language called "Pome" that has exactly 1 star on GitHub. That's not a typo - the package manager has more stars than the language it manages. It's like building a Ferrari dealership in a town with one resident.

Architecture Shenanigans

The codebase claims to support "secure package management" with "verified Git commit hashes" - because nothing screams security like trusting Git hashes from a package index that probably contains more tumbleweeds than packages. The dependency resolution system is sophisticated enough to handle complex dependency graphs, though with only one star, I suspect the most complex dependency graph it handles is "hello_world depends on stdio.h".

Virtual Environment Virtuosity

It integrates with `.pome_env` virtual environments, which is adorable. It's like watching a toddler play house - they're really committed to the bit, even though nobody else is playing. The global installation support is there because apparently someone might want to install Pome packages system-wide, presumably right next to their collection of pet rocks.

Installation Process Comedy

The installation process involves running `chmod +x install.sh` followed by `./install.sh`, which will install to `/usr/local/bin`. Nothing says "professional package manager" like a shell script that requires sudo privileges. It's like the developer read the first chapter of "Unix System Administration for Dummies" and decided that was enough security training.

Package Specification Sophistication

The `pome_pkg.json` file format is their answer to `package.json`, because apparently JSON needed yet another competing standard. The specification includes "strict anatomy" and "native extension ABI" - terms that sound impressive until you realize they're describing a package ecosystem that could fit in a tweet.

The Tragic Reality

This is a C++ implementation of a package manager for a programming language that appears to have zero adoption. It's like building a massive airport in anticipation of a city that never got built. The code quality might be excellent, but it's the software equivalent of a tree falling in a forest with no one around - does it make a sound if no one uses it?

Security Theater

The "curated index with verified Git commit hashes" is particularly rich. What's being curated, exactly? The developer's weekend projects? It's security theater at its finest - elaborate mechanisms protecting nothing from nobody.

In conclusion, Peck is a technically ambitious project that solves package management problems for a language ecosystem that exists primarily in the developer's imagination. It's either a visionary piece of infrastructure ahead of its time, or the world's most elaborate solution to a problem that doesn't exist. With 1 star, the smart money is on the latter.

Polir Complexity: 0/10

Stack:

Could not analyze deeply. Error: status_code: 429, model_name: moonshotai/kimi-k2-instruct-0905, body: {'error': {'message': 'Rate limit reached for model `moonshotai/kimi-k2-instruct-0905` in organization `org_01kd8yv9cafhtvtstgxj6r46z4` service tier `on_demand` on tokens per minute (TPM): Limit 10000, Used 9639, Requested 1138. Please try again in 4.662s. Need more tokens? Upgrade to Dev Tier today at https://console.groq.com/settings/billing', 'type': 'tokens', 'code': 'rate_limit_exceeded'}}...

linux-software-store Complexity: 6/10

Stack: Python, GTK, WebKit2, HTML, CSS, JavaScript, pkexec, Package Managers

Linux Software Store: Because Even Linux Needs an App Store

The "0 Stars, 100% Optimism" Project

Ah yes, another software store for Linux. Because what the ecosystem *really* needed was a 15th way to install Firefox. This one's special though - it's got **0 stars** and the confidence of a Silicon Valley startup. The audacity is admirable.

Architecture: Frankenstein's Monster, But Make It Modular

The project claims "modular architecture" which is Linux-speak for "we separated our HTML from our Python and called it enterprise-grade." The tech stack reads like a fever dream:

- **Python 3**: Because someone needs to glue this whole thing together
- **GTK 3**: For that authentic "Linux desktop in 2010" aesthetic
- **WebKit2**: Nothing says "lightweight" like embedding an entire browser engine
- **HTML/CSS/JS**: Making your package manager a web app, what could go wrong?

Package Manager Support: The Kitchen Sink Approach

Supporting pacman, apt, yum, dnf, AND flatpak? That's not a feature list, that's a cry for help. This codebase is about to learn why package managers are like in-laws - you can't just mash them together and expect harmony.

Security: pkexec for Maximum Drama

Using pkexec for privileged operations is like using a sledgehammer to hang a picture. Sure, it'll work, but you're one typo away from accidentally `rm -rf /`-ing someone's system. The README casually mentions this like it's no big deal. "Secure authentication" they said, as every security researcher within 50 miles felt a disturbance in the Force.

Phone-First Design: The Plot Twist Nobody Asked For

Originally designed for phone-based Linux distros? That's like designing a Formula 1 car for a go-kart track. The mental image of someone trying to install packages on a 5-inch screen using this thing is both hilarious and slightly terrifying.

Dependencies: The Real Package Manager

The installation instructions are a masterclass in dependency hell. Need GTK, WebKit2, Python GObject, AND polkit? That's not a software store, that's a full desktop environment with commitment issues.

Code Structure: Hope Over Experience

The directory structure suggests the developers *intended* to separate concerns, but we all know how that goes. Give it six months and `src/core/package_manager.py` will be a 3000-line monolith that handles everything from package installation to making coffee.

The Real MVP: The MIT License

At least they're honest about the warranty. "Use at your own risk" has never been more appropriate for a project that thinks embedding WebKit2 for a package manager is a good idea.

Verdict: Beautiful Chaos

This is either the work of a naive developer who hasn't been burned by Linux packaging yet, or a brilliant satirist creating performance art. Either way, I'm strangely proud of the ambition. You go, 0-star hero. You go.

ManhwaSearch Complexity: 6/10

`Stack: Python, Flask, Node.js, Express, Docker, JavaScript, Docker Compose, Web Scraping, Single Page Application`

Technical Architecture Breakdown

Backend (Python Flask)

The backend is your typical Flask setup with a scheduler that runs every 8 hours - because apparently manga updates are as predictable as your rent being due. The scraping system is modular, which is nice if you want to add more sites to potentially get cease-and-desist letters from.

Frontend (Node.js Express)

A classic Express server serving a SPA that's "responsive" - which in this case means it works on both your desktop and your phone while you're supposed to be working. The frontend proxies API calls to the backend, because nothing says "modern architecture" like having to explain CORS to your server.

Docker Configuration

The Docker setup is actually decent - they've got separate containers for frontend and backend, connected via Docker Compose. It's like having roommates who don't talk to each other but somehow make the rent work.

Scraping Strategy

The scraper targets mangaread.org by default, with configurable intervals and chapter limits. The "AI scraper" feature is disabled by default - probably because even the AI doesn't want to get involved in this legal gray area.

Data Management

The app stores scraped data locally, which means your manga collection survives as long as your hard drive does. No cloud, no sync, just pure self-hosted goodness - perfect for the paranoid otaku who trusts their own server more than Crunchyroll's.

Security Considerations

Let's be real - this is scraping copyrighted content. The repo has 1 star, which either means it's brand new or even the anime community has standards. The configuration allows for multiple websites, so you can diversify your sources of potential DMCA notices.

Scalability

With a default of scraping 1 chapter per manga every 8 hours, this isn't exactly built for the binge-reader. The "grab_all_chapters_favorites" option exists but comes with a warning about being "intensive" - translation: it'll probably crash your Raspberry Pi.

Development Experience

The modular architecture for scrapers is actually thoughtful - you can add new sources by inheriting from ScraperBase. It's like a plugin system, but for legally questionable content aggregation.