MEMORANDUM

To: Wakai Stakeholders
From: Dirk Harms-Merbitz
Date: Feb 14, 2026
Re: jam — A Shell for AI

A Shell for AI

The first memo described the ecosystem play. Toast for AI text transformation. iMessage for communicating with Apple's iMessage ecosystem. Each tool small, composable, written in C. But every tool in the ecosystem runs through a shell. And every shell we have is broken.

Broken how? Try to pass an LLM's output through `sed`. Try to nest quotes three levels deep. Try to write a command with a dollar sign that doesn't expand, a backslash that doesn't escape, a string that doesn't get mangled by the shell before it reaches the tool. Every developer has lost hours to this. It was tolerable when humans typed every command. It is intolerable when AI generates them.

So we built a shell called jam. Written in C. No dependencies beyond readline. Single file. What you type is what you get.

Words Are Words

Words are separated by whitespace. Quotes group words into strings — but nothing is interpreted inside them. No expansion. No escaping. No magic characters.

A dollar sign is a dollar sign. A backslash is a backslash. A string with quotes in it doesn't need quotes around its quotes. The shell is transparent plumbing. It gets out of the way.

Pipes work. Redirection works. Command chaining with `&&` and `||` works. Semicolons separate commands. Everything a working shell needs, nothing it doesn't.

Numbers are numbers

When jam sees a line that starts with a number and contains only numbers and stack operations, it does RPN math. RPN (Reverse Polish Notation) puts operators after their operands instead of between them. It works with a stack. You push numbers on, and when you hit an operator, it pops the top two numbers, does the math, and pushes the result back. No parentheses needed, no order-of-operations ambiguity. That's why engineers, programmers, scientists, finance folks love RPN calculators. RPN may be a less intuitive input method but it is genuinely more efficient once you internalize it.

```
2 3 +
5
100 2 / 3 *
150
```

The full RPN vocabulary: `+ - * / mod dup drop swap over negate abs */ .  .s cr` and `?` which pushes the exit status of the last command onto the stack. Arithmetic lives in the shell without a subprocess, without `bc`, without `awk`. Just words.

The Fallback Chain

Every line you type goes through a decision chain. Is it a builtin? Run it. Is it a RPN expression? Evaluate it. Is the first word in PATH? Fork and exec. Is it none of these? Send it to the AI.

That last step is the interesting one. jam connects directly to toastd over a Unix domain socket. Same protocol. Same authentication. Same connection pool. If toastd isn't running, the shell spawns it. No configuration. No flags.

The result: you type something the shell doesn't recognize, and instead of "command not found" you get an answer. The shell becomes the natural language interface. Not because we added a special mode. Because we made the AI the default fallback.

Type "eixt" and the AI tells you it's "exit." Type "what processes are using port 8080" and the AI tells you the command. The shell understands intent, not just syntax.

Context Is Everything

jam maintains a `.history` file. Not just for up-arrow recall — for AI context. When a line falls through to toastd, the last fifty commands go with it as part of the system prompt. The AI knows what you have been doing. It knows your working directory. It knows what you tried and what failed.

The `.history` file walks up from the current working directory, just like `.crumbs` does for toast. Put a `.history` file in a project root and that project gets its own context. Different project, different history, different AI behavior. Per-project context with zero configuration.

Environment Variables

There is no `$`. A dollar sign is a dollar sign. So jam handles environment variables the way it handles everything else: explicitly, with words.

```
set OPENAI_API_KEY sk-abc123
get OPENAI_API_KEY
sk-abc123
```

`set` calls `setenv()`. `get` calls `getenv()` and prints the value. No expansion syntax, no interpolation, no quoting around values with spaces — just words.

jam inherits all environment variables from whatever launched it. API keys set in `.bashrc` or `.zshenv` carry through. `set` is the escape hatch for per-session overrides.

Repetition

Two words handle loops. `times` for bounded repetition. `while` for unbounded.

```
5 times echo hello
```

Runs `echo hello` five times. No braces, no `do`/`done`, no syntax.

```
while make test
```

Runs `make test` until it returns a nonzero exit status. Prefix with a number for a hard cap:

```
10 while make test
```

Same loop, but stops after ten iterations regardless. Bounded and unbounded repetition, both as plain English.

Basket - The Network Bus

A shell runs on one machine. But AI agents run on many. Three linuxtoaster boxes running jam are three islands — unless they can talk to each other.

jam includes a UDP multicast bus. Two words: `send` and `listen`.

```
send status deploying
```

One UDP packet, multicast to every jam instance on the local subnet. No broker. No server. No configuration. Fire and forget.

```
listen status
web3:status deploying
```

`listen` joins the multicast group and blocks. With a key, it returns after the first match — pipeable. Without a key, it prints everything it hears. Output is `hostname:key value`.

This composes with everything else in the shell. An AI agent that monitors the network and summarizes what it hears:

```
while listen | toast "summarize this event"
```

A deploy coordinator that waits for all nodes to report ready:

```
3 times listen ready
```

The multicast group defaults to `239.255.66.85:4242`. TTL defaults to 1 — local subnet only. Override with `set JAM_MCAST_TTL 5` to cross routers. The entire message fits in a single Ethernet frame. No fragmentation. No framing. No protocol.

The topology is simple. Same shell — environment variables. Same project — shared history. Same network — multicast. Each scope is a word. Each word composes with pipes.

Scripts

jam runs scripts. Shebang line, file argument, or piped input. Comments with `#` are ignored. The `source` builtin runs a script in the current shell context — `cd` sticks, history accumulates.

```
#!/usr/local/bin/jam
# deploy script
echo deploying && make clean && make && make install
echo deployed || echo FAILED
```

No escaping gymnastics. No quoting nightmares. AI-generated scripts execute on the first try.

The Ecosystem Block

jam is a block in the linuxtoaster toolkit. Like toast, like iMessage — a C binary for communicating with Apple's iMessage ecosystem — it is written in C. Single file. Minimal dependencies. Compiles anywhere. It follows the same pattern: do one thing, do it cleanly, compose with everything.

But it occupies a special position. Every other tool runs through the shell. When the shell handles text cleanly, every pipe in the ecosystem works better. When the shell understands natural language as a fallback, the entire toolkit becomes more accessible. The shell is the foundation. We rebuilt it.

Technical Summary

Single C file. ~1,740 lines. One dependency: readline (present on every Unix system). Compiles with `make`. Installs with `make install`. Includes: tokenizer with no-escape string handling, pipe execution engine with file redirection, RPN stack interpreter for arithmetic, `set`/`get` builtins for environment variables, `N times` and `while`/`N while` for repetition, UDP multicast bus with `send`/`listen` for network-wide agent coordination, persistent command history with readline integration, direct AF_UNIX connection to toastd with autospawn, SHA-256 authentication (no libsodium — embedded implementation), `&& || ;` command chaining, `source`/`.` for script inclusion, and shebang script execution.

Conclusion

The shell is where text meets the operating system. For fifty years that boundary has been littered with escaping rules, quoting conventions, and special characters that exist only to work around the limitations of a 1970s design.

AI doesn't know those rules. It shouldn't have to. jam strips them away. What you type is what you get. What the AI generates executes on the first try. When you type something that isn't a command, the AI answers. And when you send a message, every jam on the network hears it.

The shell that doesn't escape. Put text in, get text out.

https://linuxtoaster.com