TECHNICAL GUIDE

Building a Personal AI Assistant

A guide from the builder of Woodhouse — a personal AI assistant running 24/7 on a Mac Mini M4

James Cantwell

linkedin.com/in/jamescantwell

James has spent over 25 years working in and around wealth management technology — including nearly a decade at SS&C Advent, and now running WealthTech Select where they track over 1,000 wealthtech companies. He also co-hosts the AI for Advisors podcast and recently started Ascent Growth Platforms, which will launch RIA Ascent later in 2026.

wealthtechselect.com

AI for Advisors

aiforadvisorspodcast.com

riaascent.com

OVERVIEW

What Is a Personal AI Assistant?

Not a chatbot. Not a productivity app. Something more like a chief of staff who never sleeps.

The Distinction That Matters

Most AI tools are reactive. You open a tab, type a question, get an answer, close the tab. The interaction exists in isolation from the rest of your life.

A personal AI assistant is different in one fundamental way: **it persists**. It remembers what you told it last week. It knows your meetings are tomorrow. It notices when you haven't followed up with someone. It acts on your behalf — with your permission — not just when you ask.

This changes the design problem entirely. You're not building a chatbot. You're building a trusted background process that happens to be conversational.

Three Design Principles

Ambient, not intrusive. Your assistant should feel like a capable colleague, not a notification fire hose. It speaks up when it has something worth saying. It stays quiet otherwise. The test: would you be glad you received this message if you hadn't asked for it?

Trusted actor, not suggestion engine. An assistant that can only suggest things still requires you to do the work. The goal is an assistant that can take real actions — send an email, update a CRM record, create a calendar event — within boundaries you define. Trust is earned incrementally, and every action is auditable.

Proactive, not just reactive. The reactive baseline (answer questions when asked) is table stakes. The value multiplier is proactivity: the

morning brief that synthesizes your day before you've opened your laptop, the meeting prep that runs automatically at 5am, the follow-up reminder triggered by a conversation you had three days ago.

What "24/7" Actually Means

Running always-on isn't just a deployment detail — it unlocks a qualitatively different class of behavior. A system that only runs when you interact with it can't monitor your calendar for tomorrow's meetings, can't send you a heads-up when a deal goes quiet, can't process a file that landed in your inbox overnight.

The always-on requirement shapes your infrastructure choices significantly. See Where Should It Run? for the tradeoffs between a dedicated local machine, a VPS, and a container on your dev machine.

What These Docs Cover

This guide is organized in two parts:

Core Architecture — the fundamental building blocks every personal assistant needs:

- The Runtime Skeleton: how a message flows from you to your assistant and back
- What Your Assistant Remembers: memory types, governance, and surfacing
- Building a Multi-Agent System: routing, personas, and model selection
- Moving from Reactive to Proactive: scheduled and event-triggered behavior
- Taking Real-World Actions Safely: tiers, confirmations, and audit trails
- Connecting to Your World: CRM, calendar, email, notes
- Adding Phone and Voice Capability: when voice is worth the complexity

Design Questions — short opinionated takes on the decisions you'll face early:

- What should it remember?
- Reactive or proactive?
- How many agents?
- How much should I trust it?
- Which interface?
- Where should it run?

What These Docs Are Not

This is not a tutorial. There's no step 1, step 2, clone this repo. The goal is to give you a clear mental model of the architecture, design decisions, and tradeoffs — so you can build something that fits your life, your stack, and your preferences.

The examples draw from a real system (Woodhouse, a personal assistant running on a Mac Mini M4). But the patterns apply regardless of whether you use the same tools.

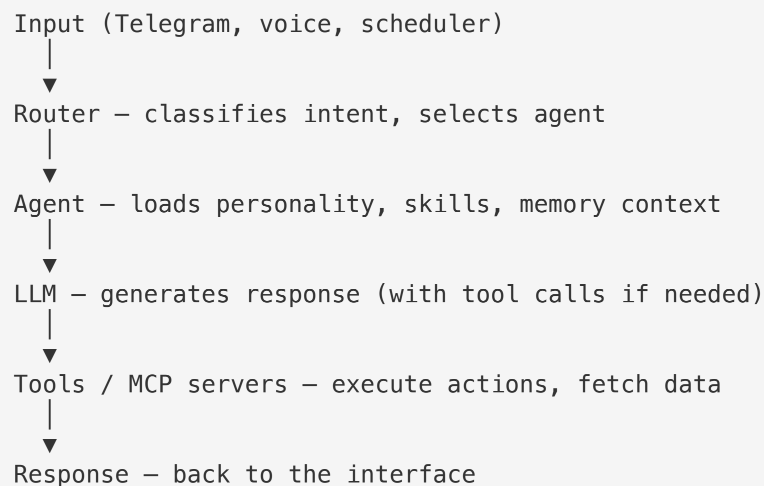
ARCHITECTURE

The Runtime Skeleton

Every message — whether from you or from the scheduler — flows through the same pipeline. Understanding this loop is the foundation for everything else.

The Core Request Loop

At its simplest, a personal AI assistant does one thing: it takes an input, decides how to handle it, calls a language model with the right context, and returns a response.



```
Input (Telegram, voice, scheduler)
  │
  ▼
Router — classifies intent, selects agent
  │
  ▼
Agent — loads personality, skills, memory context
  │
  ▼
LLM — generates response (with tool calls if needed)
  │
  ▼
Tools / MCP servers — execute actions, fetch data
  │
  ▼
Response — back to the interface
```

This loop handles both **reactive** messages (you send something, it replies) and **proactive** messages (the scheduler triggers it, and the response goes to you unprompted). The pipeline is identical — only the trigger differs.

Key Architectural Decisions

Single process vs. microservices. Start with a single process. One Node.js or Python process that owns the Telegram bot, the scheduler, the orchestrator, and the action layer is far easier to debug and deploy than a

distributed system. Split only when you have a concrete reason — for example, a voice server with strict latency requirements benefits from being isolated, but the rest of the system does not.

Where state lives. Session state (the last few messages in a conversation) lives in memory. Long-term memory (facts, preferences, learned patterns) lives in a database — Postgres, SQLite, or even a well-structured file system. The interface between short-term and long-term is a memory layer that decides what gets persisted and what gets injected into the next context. See What Your Assistant Remembers.

Session continuity. Conversational context doesn't reset between messages unless you explicitly clear it. A sticky agent window (e.g., 30 minutes of inactivity before resetting) gives the assistant short-term memory without requiring you to re-explain context on every message.

LLM as the reasoning layer only. The LLM decides what to do. Separate systems execute actions, manage state, and handle I/O. This separation is what makes the action layer safe (see Taking Real-World Actions Safely) and what makes the memory layer reliable (the LLM doesn't directly write to your database — your code does, after verifying the intent).

The Role of Tools

Language models can't do anything on their own except generate text. The "hands" of your assistant are tools — functions the LLM can invoke during its reasoning process.

Tools fall into two categories:

Read tools — fetch data to inform responses: calendar events, CRM contacts, email summaries, task lists, web search. These are safe to execute automatically.

Write tools — take actions in the world: send an email, update a record, create an event, run a shell command. These require more care. See Taking Real-World Actions Safely.

The Model Context Protocol (MCP) is a standardized interface for connecting tools to LLMs. Whether you use MCP or build your own tool interface, the pattern is the same: the LLM describes an intent (`send_email(to=..., subject=..., body=...)`), your code validates it, and your code executes it.

Proactive Messages in the Same Loop

The scheduler is just another input source. A cron job fires at 7am, constructs a prompt ("prepare the morning brief for today"), routes it through the same orchestrator, and the response goes to Telegram instead of back to an API caller.

This means you don't need a separate "proactive system" — you need a scheduler that knows how to construct the right prompt and where to send the output. See Moving from Reactive to Proactive.

Connecting to Other Docs

- Message routing and agent selection → Building a Multi-Agent System
- Memory injection and persistence → What Your Assistant Remembers
- Tool execution and confirmation → Taking Real-World Actions Safely
- Scheduled triggers → Moving from Reactive to Proactive
- External integrations → Connecting to Your World

MEMORY

What Your Assistant Remembers

Memory is what separates a personal assistant from a stateless chatbot. Getting it right means knowing what to store, how long to keep it, and when to surface it.

Four Types of Memory

Session context is the short-term working memory of a conversation — the last N messages exchanged. It's held in RAM, disappears when the session resets, and tells the LLM what was just discussed. This is the minimum viable memory system and the default behavior of any LLM API.

Long-term facts are things you tell your assistant that should persist indefinitely: your preferred way to communicate, key people in your life, standing priorities, background context about your work. These are stored in a database and injected into relevant system prompts.

Learned preferences are patterns the assistant infers over time: that you prefer bullet points over paragraphs, that you want CRM updates after every call, that 7am briefings are too early on Mondays. Unlike explicit facts, these are derived from repeated behavior and feedback signals.

Action history is a log of everything the assistant has done on your behalf — emails sent, tasks created, CRM records updated. This serves two purposes: auditing (what happened?) and context injection (the assistant knows it already sent that follow-up).

Short-Term vs. Long-Term

The boundary is simpler than it sounds: **if you'd want the assistant to know it next week, it's long-term**.

Short-term memory (session context) handles the immediate conversation. Long-term memory handles everything else. The failure

mode to avoid is treating everything as short-term — your assistant forgets who your most important client is between sessions — or treating everything as long-term, which creates noise in every prompt.

A practical starting point:

- Session context: last 20–30 messages, auto-cleared after 30 minutes of inactivity
- Long-term facts: explicit statements tagged as persistent ("remember that…"), plus structured records from integrations (CRM contacts, calendar events, tasks)
- Learned preferences: initially manual, automated later when you have enough signal

---

How Memory Surfaces

Memory can be surfaced two ways:

Proactive injection — relevant facts are automatically included in the system prompt before the LLM sees the message. A morning briefing automatically gets today's calendar events. A CRM-related message automatically gets the relevant contact's history. This requires a retrieval step that matches the incoming message to stored context.

On-demand retrieval — the LLM can call a memory search tool during its reasoning. Useful for one-off lookups ("what did we decide about the pricing model?") where you don't want to inject everything upfront.

In practice, you use both: inject the high-confidence relevant context automatically, and allow the LLM to search for more if needed.

Memory Governance

Memory that isn't managed becomes a liability. A few principles:

Decay matters. Some facts are time-sensitive. A person's job title, a deal's status, a project's priority — these change. Build in a way to mark memories as stale or override them.

Explicit over inferred. Start with explicit memory (things you consciously tell the assistant to remember) before building inference. It's easier to debug and more trustworthy.

Summarization at scale. As conversation history grows, summarizing older sessions into compressed context keeps token usage manageable without losing the thread.

Deletion is a feature. Your assistant should be able to forget things. A `/forget` command or equivalent isn't just nice to have — it's a trust mechanism.

Stack-Agnostic Patterns

The memory architecture described here doesn't require any specific database. What matters:

- A key-value or row-based store for long-term facts and preferences (Postgres, SQLite, Airtable, even a JSON file for a simple system)
- A vector store for semantic retrieval (pgvector, Pinecone, Chroma, or any embedding-capable database) — only necessary once you have enough content that keyword search breaks down
- A logging table for action history

Start simple. A flat table of `(key, value, timestamp)` facts gets you surprisingly far. Add semantic search when you actually need it.

AGENT DESIGN

Building a Multi-Agent System

One general-purpose agent is the right place to start. A multi-agent system is what you evolve into when one agent starts breaking down. Understanding why it breaks — and how routing works — is what this doc covers.

Why One Agent Eventually Breaks Down

A single agent with a long, catch-all system prompt has a coherence problem. The LLM is simultaneously a developer, a sales strategist, a calendar assistant, and a personal trainer. The prompt becomes bloated. The model gets confused about which mode it's in. Responses lose sharpness.

The second problem is cost and speed. Routing a "what's on my calendar today?" message through a 200K-context Opus model is wasteful. Some tasks need deep reasoning; most don't.

The third problem is context window pollution. A CRM-aware agent needs Pipedrive context injected. A coding agent needs file system access and project context. Giving everything to every request bloats the context, costs tokens, and degrades focus.

Multi-agent systems solve all three: each agent has a tightly scoped system prompt, the right model tier, and only the context it needs.

What a "Agent" Actually Is

In this architecture, an agent is:

1. **A system prompt** — the agent's personality, scope, and instructions
2. **A model** — which LLM to use (and which tier: fast/cheap vs. slow/capable)
3. **A tool policy** — which MCP servers and tools the agent can access

4. **A timeout** — how long it can run before the framework cuts it off

That's it. An agent isn't a separate process or service. It's a configuration bundle that the orchestrator selects and applies before calling the LLM.

Routing: How the Right Agent Gets Selected

Routing is the orchestrator's job. It looks at an incoming message and decides which agent handles it. A working routing strategy has layers:

Pattern matching — fast, cheap intent classification. Keywords, regex, or a lightweight LLM call that bins the message into a category (CRM, calendar, coding, personal, etc.) and maps it to an agent.

Sticky routing — once an agent responds, it stays active for a time window (e.g., 30 minutes). Follow-up messages go to the same agent without re-classification. This preserves conversational context and feels more natural.

Explicit override — the user can lock to a specific agent (`/agent developer`), bypassing routing entirely. Useful when you know what you need.

Handoffs — an agent can suggest that a different agent would handle the next message better. Rather than silently switching, the interface surfaces this as an option the user can accept or decline.

The priority order matters: explicit override > sticky agent > pattern matching > default.

Personality Files

System prompts shouldn't live in code. Keep them in files — one file per agent, loaded at startup. This makes them easy to edit, version-control, and experiment with without touching code.

A personality file typically contains:

- The agent's name and role

- Tone and communication style
- Standing instructions (always check CRM before replying about a contact)
- Scope boundaries (what this agent handles and what it should hand off)
- Any static context (e.g., the company's current priorities)

Skills — reusable prompt fragments for specific domains — can be conditionally injected on top of the personality. A CRM agent might always have the sales playbook injected; a coding agent gets coding standards.

Model Selection Strategy

Not all messages need the same model. A tiered approach:

Tier	Use for	Example
Fast/small (Haiku-class)	Greetings, quick lookups, yes/no	"What time is my next meeting?"
Mid-tier (Sonnet-class)	Most tasks, tool use, multi-step reasoning	CRM updates, drafts, research
Top-tier (Opus-class)	Strategic synthesis, complex architecture, investor prep	Board prep, deep strategy

The 90/9/1 rule is a reasonable starting point: ~90% of requests go to mid-tier, ~9% to fast, ~1% to top-tier. Tune based on your actual usage patterns and cost tolerance.

Per-agent model defaults, with user override for one-shot escalation (`/model opus`), gives you flexibility without complexity.

See Also

- How many agents? — the decision question version of this doc
- The Runtime Skeleton — where routing fits in the overall loop

PROACTIVE SCHEDULER

Moving from Reactive to Proactive

The reactive baseline — answer when asked — is useful. The proactive layer is where a personal assistant earns its place.

Three Trigger Types

Proactive behavior is triggered three ways:

Time-triggered — cron-style schedules. A morning brief at 7am on weekdays. A weekly pipeline review on Monday morning. A Friday content summary. These are predictable, easy to implement, and the right starting point.

Event-triggered — something happens and the assistant responds. A meeting just ended → extract tasks and follow-ups. A file landed in the downloads folder → process it. A calendar event was created → check if prep is needed. These require either polling or a webhook/watcher architecture.

Data-triggered — a condition in your data is met. A deal has been quiet for 14 days → send a follow-up prompt. An investor was marked "warm" 30 days ago → check in. These are the most powerful and the hardest to get right — they require knowing your data well enough to write good trigger conditions.

Start with time-triggered. It requires zero additional infrastructure and delivers immediate value.

The Morning Brief Pattern

A morning briefing is the highest-return proactive behavior to build first. It synthesizes — not summarizes — the day ahead.

The distinction matters: a summary is a list of your calendar events. A synthesis is "you have a board presentation at 2pm where the main concern last time was burn rate, followed by a call with your largest client who mentioned they were evaluating alternatives." One is a data dump; the other is useful.

What makes a good briefing:

- **Top priorities** — not everything, the 2-3 things that actually matter today
- **Calendar context** — not just event names, but relevant background from memory and integrations
- **Anything that needs attention** — stale follow-ups, waiting items, flagged tasks

What makes a bad briefing: everything. If it's long, it gets skimmed. If it gets skimmed, it stops being useful. Ruthlessly filter.

Meeting Prep and Post-Meeting Tasks

Pre-meeting prep is a natural fit for proactive behavior. Thirty minutes before a meeting, pull the relevant CRM records, recent email threads, and any notes from the last time you spoke with these people. Surface it without being asked.

The trigger is straightforward: scan the calendar for meetings starting in the next N hours that meet some qualifying criteria (external meetings, certain meeting types), and run a prep pipeline for each one.

Post-meeting task extraction closes the loop. When a meeting ends, pull the transcript (if you have a transcription integration), extract action items, and route them back to you for confirmation before creating tasks. The "route back for confirmation" step matters — automated task creation without review creates noise faster than it creates value.

Avoiding Notification Fatigue

The failure mode for proactive assistants is becoming another notification source. Some safeguards:

Relevance filters. Before sending a proactive message, ask: would the user want this right now? A stale deal alert at 11pm is noise. The same alert at 9am is useful.

Quiet hours. Don't send proactive messages outside working hours unless explicitly configured otherwise.

Batching. A single morning briefing is better than five separate notifications throughout the morning. When possible, aggregate related signals into one message.

Suppression logic. If the user is in a meeting, delay. If they're traveling, adjust. Awareness of context makes the difference between helpful and intrusive.

The Same Pipeline

The key architectural insight: proactive jobs don't need their own system. They construct a prompt, pass it to the orchestrator, and the response goes to the interface. The agent selection, memory injection, tool access — all of it works exactly as it does for reactive messages.

This means you can build a fully proactive system without any new infrastructure once you have a working reactive pipeline. The scheduler is just an additional input source.

See Also

- Reactive or proactive? — the decision version of this doc
- The Runtime Skeleton — how proactive triggers fit the core loop

ACTION LAYER

Taking Real-World Actions Safely

Letting your assistant take actions on your behalf is where the real leverage is. It's also where the real risk is. An action layer makes both manageable.

Why a Separate Action Layer

The LLM decides what to do. A separate layer executes it. This separation is not optional — it's the mechanism that makes autonomous action safe.

Without it, the LLM has direct access to your systems. A confused prompt, a misunderstood intent, or a subtly wrong tool call can trigger real consequences: an email sent to the wrong person, a file deleted, a calendar event created at 3am. With it, every action passes through a layer that knows what's reversible, what requires confirmation, and what should never be automated.

The action layer is also where audit trails live. Every action that runs — whether automatic or confirmed — gets logged with a timestamp, the agent that requested it, and the outcome. When something goes wrong (and it will), this log is how you figure out what happened.

The Tier Model

Actions can be tiered by their risk profile:

Tier 0 — Read operations. Getting clipboard content, checking battery level, reading a file, fetching a calendar event. No side effects. Execute immediately, no confirmation needed.

Tier 1 — Local write operations. Creating a note, opening a URL, setting a volume level, sending a desktop notification. Low reversibility concern. Execute and log, no confirmation needed.

Tier 2 — Send or delete operations. Sending an iMessage, deleting a file, emptying trash, posting to a service. These have external effects or can't be undone. Require explicit confirmation before executing.

Tier 3 — System operations. Restarting a service, modifying system configuration, actions with broad blast radius. Always confirm, even if the user previously said "just do it."

The right tier for any action is determined by one question: **how bad is it if this runs when it shouldn't?**

Confirmation Flows

For Tier 2 and 3 actions, the flow is:

1. The LLM generates an action request
2. The action layer intercepts it and sends a confirmation message to the user interface
3. The user approves or declines (inline button in Telegram, for example)
4. The action layer executes or cancels accordingly

The confirmation message should be specific: not "send a message?" but "Send to James: 'Following up on the proposal — are you still interested?'"

Vague confirmations get auto-approved by habit, defeating the purpose.

The Allowlist Philosophy

Shell commands and system actions should operate on an allowlist, not a blocklist. The assistant can run the commands you have explicitly approved — nothing else. Adding a new command requires a code change, not a prompt change.

This sounds restrictive. It is, intentionally. An LLM that can run arbitrary shell commands is an LLM that can be prompted — accidentally or otherwise — into running arbitrary shell commands. The allowlist makes the blast radius of any mistake bounded and predictable.

The same principle applies to Apple Shortcuts, external API calls, and any other action surface: define the approved actions explicitly, and reject everything else.

Audit Trails

Every action should produce a log entry:

```
timestamp  | agent      | action           | parameters
| status   | confirmed_by
2026-03-15 | chief-staff| send-imessage    | {to:
"James", body: …}  | success | user
2026-03-15 | scheduler  | create-note      | {title:
"Meeting recap"}| success | auto
```

This log answers "what did it do?" after the fact. It also builds trust over time — being able to review a week of actions and see that everything was sensible is how you get comfortable extending more autonomy.

See Also

- How much should I trust it? — the decision version of this doc
- The Runtime Skeleton — where the action layer fits in the overall loop

INTEGRATIONS

Connecting to Your World

Your assistant is only as useful as the data it can see and the systems it can touch. Integrations are how you close that gap.

The Integration Philosophy

Your assistant should live in your data, not the other way around. The goal is not to build a new system that everything syncs into — it's to give your assistant read and write access to the systems you already use.

This means: your calendar stays in your calendar. Your email stays in your email client. Your CRM stays in your CRM. The assistant connects to all of them, but doesn't own any of them.

The practical benefit is resilience. If the assistant goes down, your data is untouched. If you switch CRMs, you update one connection, not a data model.

Integration Categories

Calendar and email are the highest-value integrations to build first. Calendar context is essential for meeting prep and morning briefings. Email access enables inbox triage, draft generation, and follow-up tracking. Both are read-heavy with selective write operations (create event, send email).

CRM is where relationship context lives. Knowing who you talked to, when, and what was discussed turns generic responses into personalized ones. Even a lightweight CRM integration — just reading contact and deal records — significantly improves the quality of sales and relationship-related assistance.

Notes and knowledge base give your assistant access to your thinking. Meeting notes, project documents, research, drafts — if your assistant

can search and read these, it can answer questions about your own work rather than just about public knowledge.

Communication platforms (Slack, Teams, SMS) are primarily used as the assistant's output channel — where it sends you messages and confirmations. Bidirectional access (reading your Slack messages) is valuable but adds complexity; defer until the basics are working.

Files — monitoring a downloads folder, processing documents that arrive via AirDrop or email attachment, writing meeting notes to your notes app — close the loop between the physical and digital worlds.

MCP as a Pattern

The Model Context Protocol (MCP) is a standardized way to expose tools and data to LLMs. The key benefit isn't the protocol itself — it's the pattern: each integration is a separate, scoped server that the orchestrator connects to. Adding a new integration means adding a new server, not modifying the core system.

Whether you use MCP or build your own tool interface, apply the same principle: **integrations are modular and independently replaceable**. Your assistant shouldn't need to know that you switched from Google Calendar to Outlook — that's the integration layer's problem.

Read vs. Write Permissions

Design each integration with explicit permission tiers:

Integration	Read	Write
Calendar	Always on	Confirm before creating/modifying events
Email	Always on	Confirm before sending; auto-draft is fine
CRM	Always on	Auto-log calls/notes; confirm deal stage changes
Files	Path-allowlisted	Confirm before delete; auto-create is fine

Integration	Read	Write
Shell	Allowlisted commands only	Always confirm for new commands

Read permissions are low-risk and can be granted broadly. Write permissions should be conservative and expanded incrementally as trust is established.

Graceful Degradation

Every integration will fail at some point — an OAuth token expires, an API is down, a network is unavailable. Your assistant should degrade gracefully rather than failing completely.

A few patterns:

- **Fail informatively.** If calendar data isn't available, say so rather than generating a briefing with no calendar context.
- **Cache aggressively.** For read integrations, cache recent data so a brief API outage doesn't break everything.
- **Fail per-integration, not globally.** A broken CRM connection shouldn't prevent the assistant from answering email questions.
- **Surface auth issues.** When a token expires, notify the user rather than silently returning empty results.

VOICE

Adding Phone and Voice Capability

Voice is a qualitatively different interface — not just text input via microphone, but a different interaction model entirely. Build it last, and only if it fits how you actually want to interact with your assistant.

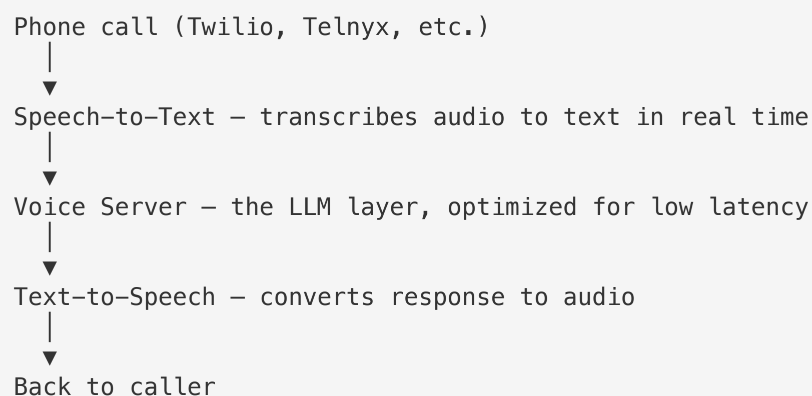
Why Voice Is Architecturally Different

Text interaction is asynchronous and forgiving. You can take five seconds to think, edit your message, and send. The assistant can take ten seconds to respond and you won't notice.

Voice is synchronous and unforgiving. Latency above two seconds feels broken. There's no markdown — no bullet points, no headers, no code blocks. Responses must be spoken aloud, which means they need to be structured for listening, not reading. Turn-taking must be handled explicitly (who's speaking? when can the other party interrupt?).

These constraints change almost every design decision: model choice, response length, tool use, and the pipeline architecture itself.

The Voice Call Pipeline



Each stage adds latency. The total budget for a natural-feeling exchange is roughly 1–2 seconds end-to-end. This constrains every choice:

- **STT**: streaming transcription (words processed as they're spoken, not after silence) shaves hundreds of milliseconds
- **LLM**: smaller models and shorter responses over maximum quality
- **TTS**: pre-caching common responses, streaming audio output before the full response is generated

---

Synchronous vs. Asynchronous Tool Use

Tool use during a voice call requires careful handling. If your assistant needs to check your calendar to answer "what's on my schedule tomorrow?", that lookup must complete synchronously — while you're on the call, in real time.

For fast, read-only tools (calendar, tasks, recent emails, CRM contacts), synchronous execution works. Design these tools to return quickly and structure the assistant's response to account for the latency.

For slow or write operations (send an email, update a CRM record, create a task), use a fire-and-forget pattern: acknowledge the request during the call ("I'll send that after the call"), then execute the action asynchronously and confirm via your text interface (Telegram, etc.) when done.

The distinction: **reads happen during the call, writes happen after**.

When Voice Is Worth the Complexity

Voice adds significant architectural complexity. It's worth building when:

- You interact with your assistant while doing other things (driving, cooking, walking)
- The speed of spoken interaction meaningfully outweighs typing
- You want your assistant to be reachable from any phone without opening an app

It's probably not worth it (yet) if:

- Your primary use cases are long-form drafting, code review, or anything that produces structured output
- You're still building out the core text-based system
- You don't have a consistent latency budget to work within

The voice layer is a capability extension on top of a working core system — not a replacement for it, and not a good first step.

Latency Tradeoffs

The latency dial has three settings:

Model size. A faster, smaller model responds in 300ms. A slower, larger model responds in 2–3 seconds. For voice, the smaller model wins almost every time — the interaction model compensates for slightly lower capability.

Streaming vs. non-streaming. Streaming LLM output (sending audio as tokens are generated rather than waiting for the full response) can cut perceived latency in half. Worth the complexity in a voice context.

Tool use detection. To know whether a response requires tool use, you often need to wait for the LLM's first response. One approach: make the first API call non-streaming to detect tool use, then stream the final response. This adds a small fixed overhead but enables synchronous tool execution.

The right answer is: profile your actual call experience, find the dominant source of latency, and optimize that specifically.

SUMMARY

Where to Go From Here

You've covered the full architecture of a personal AI assistant — from the runtime skeleton to voice calling. This is the closing chapter.

What You Now Know

If you've read through these docs, you have a working mental model of every major component:

- **The runtime loop** — every message, reactive or proactive, flows through the same interface → router → agent → LLM → response pipeline
- **Memory** — four types with different lifespans, surfaced by injection or retrieval, governed by decay and explicit deletion
- **Agents** — configuration bundles, not separate services; routing layers that preserve context and allow escalation
- **The proactive layer** — time, event, and data triggers that all feed the same pipeline; the morning brief as the highest-return starting point
- **The action layer** — tiered by reversibility, allowlisted not blocklisted, every action audited
- **Integrations** — your assistant lives in your data, not the other way around; modular, independently replaceable connections
- **Voice** — a capability extension, not a foundation; synchronous reads, asynchronous writes, latency measured in milliseconds

And you've worked through the foundational decisions: what to remember, how proactive to be, how many agents, how much autonomy to extend, which interface to use, and where to run it.

What to Build First

If you're starting from scratch, the order matters:

1. **Interface + reactive core** — get a message in, get a response out. Telegram bot → orchestrator → one agent → LLM. Nothing else.
2. **Memory** — add session continuity and one long-term fact store. Test that context persists across sessions.
3. **One proactive job** — the morning brief. Time-triggered, synthesizes your calendar. Validate that it's actually useful before building more.
4. **Integrations** — connect calendar and email first. CRM second if it's relevant to your work.
5. **Action layer** — start with Tier 0 and Tier 1 only. Introduce confirmation flows for Tier 2 once you trust the baseline.
6. **More agents** — only when you hit the coherence or cost problems described in the agent design doc.
7. **Voice** — last, and only if phone interaction genuinely fits how you work.

Each step produces something useful on its own. Don't skip ahead.

What Makes This Different From a Chatbot

The gap between a chatbot and a personal assistant isn't the model. It's three things:

Persistence. It remembers. Not just the last message — your priorities, your relationships, your preferences, your history. This is what makes responses feel like they come from someone who knows you, not someone who just met you.

Proactivity. It shows up. You don't have to remember to ask. The morning brief is there before you open your laptop. The meeting prep is ready thirty minutes before the call. The follow-up prompt arrives two weeks after the conversation you forgot about.

Agency. It acts. Not just suggestions — real actions, within boundaries you define, with a full audit trail. This is where the compounding leverage comes from: not just answering questions, but closing loops.

These three things are what you've been building toward. The architecture exists to support them reliably, safely, and at the pace you're comfortable with.

Follow Woodhouse

These docs were written from the experience of building Woodhouse — a personal AI assistant running 24/7 on a Mac Mini M4. Woodhouse handles everything from morning briefings and meeting prep to CRM updates, investor pipeline reviews, and phone calls.

If you're building something similar and want to follow along with what's being learned, added, and changed, visit The Woodhouse Website.

Good luck. The first version doesn't have to be impressive — it just has to be yours.

DESIGN QUESTION — WHAT TO REMEMBER

What Should It Remember?

The temptation is to remember everything. It feels safe — you can't miss something if you stored it all. In practice, an assistant that remembers everything is an assistant that surfaces noise as often as signal.

The better question is: **what would you tell a new chief of staff on their first day?**

You'd tell them your most important clients and the current state of those relationships. You'd tell them your standing priorities and how you like to communicate. You'd tell them the context behind the projects you're working on. You wouldn't read them your entire email history.

That's the filter. If it's the kind of thing you'd orient a new person with, it's worth remembering. If it's transactional detail that would be noise the next time it's surfaced, it probably isn't.

Categories Worth Storing

People context. Who matters to you, your relationship with them, the last significant interaction, and anything you specifically want to remember about them. This is the highest-value memory your assistant can have — it's what makes responses feel personal rather than generic.

Standing preferences. How you like to communicate, what formats you prefer, recurring decisions you've made (always BCC this address, always add agenda items to this calendar). Things you've told the assistant explicitly and don't want to repeat.

Project and goal context. The background behind the work you're doing. Not the tasks (those belong in a task system) but the "why" and "what we decided" — the things that would otherwise require you to re-explain context every session.

Action history. What the assistant has done on your behalf. Not as detailed logs (those are for auditing) but as a summary layer: "sent the

follow-up to James last Thursday", "created the board deck outline last week."

What Decays

Some memory has a shelf life:

- Deal stages and pipeline status change frequently — don't treat CRM state as stable long-term memory; query it fresh
- Meeting outcomes are useful for a week or two, then become noise
- Task lists belong in a task system, not in general memory

The Heuristic

If you'd repeat it in a future conversation with your assistant — if you'd start a new session by saying "by the way, remember that…" — then it's worth storing. If it's context that was only relevant to the conversation you just had, let it go.

When in doubt, err toward storing and add a decay mechanism later. Missing context is more painful than occasional noise.

DESIGN QUESTION — REACTIVE VS PROACTIVE

Reactive or Proactive?

Build reactive first. Always. A proactive system that isn't grounded in a solid reactive foundation becomes noise.

But once the reactive baseline works — your assistant reliably handles what you ask — the move to proactive is where the real leverage is.

The Reactive Ceiling

Reactive interaction has a hard ceiling: the assistant can only be as useful as the things you remember to ask. You remember to ask your assistant to prepare for a meeting when you're already stressed about it five minutes before it starts. You remember to ask for a deal review when a deal has already gone cold. The assistant is always one step behind.

Proactive behavior breaks that ceiling. Your assistant doesn't wait for you to ask — it notices that the meeting is tomorrow, runs the prep this morning, and delivers it when you're still in a position to use it.

The Test

Before building any proactive behavior, apply this test: **if I received this message without asking for it, would I be glad I got it?**

If the answer is "usually yes," it belongs in your proactive system. If the answer is "sometimes," build in a relevance filter. If the answer is "probably not," it's not a proactive behavior — it's a notification, which is different and usually worse.

Starting Small

The highest-return first proactive behavior is a morning brief. One message, every weekday morning, synthesizing your day. It's time-triggered (simple to implement), high-signal (you already know what would be useful), and self-contained (failure doesn't break anything else).

Build that. Get it right. See how it changes your mornings. Then, based on what you actually notice yourself wanting, extend from there.

Don't build five proactive behaviors before you've validated one. Proactive noise is worse than no proactive behavior at all — it trains you to ignore your assistant.

The Spectrum

Proactive behavior exists on a spectrum from passive to active:

- **Passive**: surfaces information you'd want ("here's your day")
- **Semi-active**: suggests actions ("you haven't followed up with James in two weeks — want me to draft something?")
- **Active**: takes actions on your behalf without asking ("sent your standard follow-up to James")

Start passive. Move right only when the left side has earned your trust and is delivering consistent value.

DESIGN QUESTION — HOW MANY AGENTS

How Many Agents?

Start with one.

This is not a hedge — it's a design principle. A single agent with a well-crafted system prompt handles the overwhelming majority of use cases better than a hastily partitioned multi-agent system. The cost of routing complexity, handoff failures, and split context is real. Pay it only when you have a concrete reason to.

When One Agent Breaks Down

The signal that you need more than one agent is not "I have a lot of use cases." It's specific failure modes:

Context window incoherence. Your system prompt is trying to be everything to everyone and the responses have lost focus. The agent doesn't know whether to respond like a CRM expert or a coding assistant.

Cost inefficiency. You're routing routine questions ("what's on my calendar?") through a high-capability model when a smaller model would do fine. Model selection per-agent fixes this.

Context pollution. Every request is loading CRM context, coding context, and calendar context regardless of what the message is actually about. This wastes tokens and degrades the signal-to-noise ratio of the context.

If you're not experiencing these problems, you don't need multiple agents yet.

The Case for Specialization

When you do split, split along natural task boundaries with meaningfully different context needs:

- A **research agent** that gets web search access and no CRM context

- A **CRM agent** that gets your full relationship and deal context
- A **developer agent** with filesystem access and coding standards injected
- A **quick-response agent** (tiny model, no tools) for greetings and trivial lookups

Each agent has a tighter, more coherent prompt, the right model for the job, and only the tools and context it actually needs.

Warning Signs

You've added too many agents when:

- Routing failures become a regular complaint ("it sent this to the wrong agent again")
- You spend more time tuning routing rules than building useful behavior
- Every conversation starts with the user specifying which agent they want

The right number of agents is the minimum needed to solve your actual problems. For most people, that's 3–5 well-scoped agents, not 15.

DESIGN QUESTION — ACTION TRUST

How Much Should I Trust It?

The honest answer: less than you think at first, more than you think after six months.

Trust in an AI assistant isn't binary and it isn't static. It's a spectrum, and you move along it based on evidence — specifically, the track record of the actions your assistant has taken on your behalf.

The Spectrum

At one end: **suggest only**. The assistant generates recommendations, you execute them. No autonomous action whatsoever. This is safe, auditable, and exhausting — you become the bottleneck for everything.

At the other end: **fully autonomous**. The assistant acts on its interpretation of your intent without confirmation. This is fast, leveraged, and occasionally catastrophic.

The right point on this spectrum depends on two things: the reversibility of the action, and the quality of your assistant's track record on similar actions.

Irreversibility Determines Tier

This is the cleanest mental model for action trust: **the harder an action is to undo, the more confirmation it needs.**

Reading your calendar: fully automatic. You can't un-read a calendar event.

Creating a task: automatic, but logged. Low stakes, easy to delete.

Sending an email: confirm first. Emails can't be unsent.

Deleting a file: confirm first, and log the path so you can recover from backup.

Posting publicly: always confirm, regardless of your general trust level.

How Trust Is Earned

The mechanism is simple: look at the audit trail. After a week, review what your assistant did automatically. Were those decisions right? Were there any surprises?

If the answer is consistently "yes, that was right," extend automatic approval to that category. If there were surprises, add a confirmation step.

This is how you move from "confirm all Tier 1 actions" to "auto-approve Tier 1 actions" — not by deciding in the abstract that you trust the system, but by observing that it has made good decisions in practice.

One Rule That Saves You

Whatever trust level you settle on: **keep the audit trail**. Even for actions you've fully automated, log them. When something eventually goes wrong (and something will), the log is how you diagnose it, and how you rebuild confidence after fixing it.

Trust without observability isn't trust — it's hope.

DESIGN QUESTION — INTERFACE CHOICE

Which Interface?

The interface question matters more than most builders expect. Your choice determines how often you actually use your assistant, what kinds of interactions feel natural, and what's technically possible.

The principle: **meet your assistant where you already are**.

Why Telegram Works Well

Telegram has several properties that make it well-suited as a primary interface for a personal AI assistant:

Cross-platform by default. Same experience on iOS, Android, Mac, Windows, and web — without building any of them yourself. Your assistant is accessible from every device you own.

Persistent thread. The conversation with your assistant is a single, scrollable thread. Context doesn't disappear when you close the app. You can reference what was said two weeks ago.

Voice-native. Voice messages are first-class in Telegram. You can send a voice note and get a text reply, or ask for an audio response. This enables voice interaction without a separate phone infrastructure.

Notification-first. Proactive messages — your morning brief, meeting prep, follow-up alerts — land in the same thread as your conversations. There's no separate channel to monitor.

Bots are free and unrestricted. Unlike Slack (which has significant rate limits and API restrictions for bots) or WhatsApp (which requires a business account and has strict policies), Telegram's bot API is permissive, well-documented, and free.

Alternatives and Their Tradeoffs

SMS — universally accessible, no app required, works on any phone. The constraints: no message history in any useful form, very limited

formatting, no inline buttons or interactive elements, rate limits from carriers.

Slack — good if your work life already lives there. Real downside: Slack's bot API has rate limits that make high-frequency proactive messaging painful, and the cost to operate a bot at scale is non-trivial.

Discord — similar to Telegram in openness, better for community/group contexts, less natural as a personal assistant interface.

Custom app — maximum control over UX, zero constraints. Also: you have to build and maintain it. Probably not worth it until you've validated your assistant with an existing platform.

Web interface — easy to build, accessible from a browser. Loses the mobile notification model, which is where most of the proactive value lives.

The Right Question

Don't ask "what's the best interface?" Ask "where do I already go when I want to communicate quickly?" For most people, that's a messaging app on their phone. Pick the messaging app with the most capable bot API. For most builders, that's Telegram.

DESIGN QUESTION — WHERE TO RUN

Where Should It Run?

This is one of the first decisions you'll make and one of the ones with the most downstream consequences. Three options: a dedicated local machine, a VPS in the cloud, or a container on your development machine.

Option 1: Dedicated Local Machine

A spare Mac Mini, an old laptop, or any always-on computer on your home network.

The case for it:

Always-on without cloud costs. Once the hardware is paid for, the incremental cost of running the assistant is electricity — roughly $3–5/month for a Mac Mini. There's no per-hour instance cost, no egress fees, no surprise billing.

Native Mac capabilities. If you're on Apple hardware, you get access to things that simply aren't available in a cloud VM: osascript for controlling macOS, Apple Shortcuts, the local filesystem without network overhead, AirDrop, native notification center, iCloud access. These capabilities meaningfully expand what your assistant can do.

Security posture by default. No public-facing ports. Remote access via a tool like Tailscale means the machine is on your private network, not the internet. Credentials never leave the machine. The attack surface is the Tailscale network, not the open web.

Developer experience. Logs are local. Debugging is `tail -f`. Deploying a change is saving a file and restarting the process. There's no deploy pipeline, no container registry, no rollback strategy needed. You are physically close to the machine.

The tradeoff: you depend on a physical machine. If it loses power, you lose the assistant. If you travel and forget to leave it on, same. For most

people in a home environment with reasonable uptime, this isn't a real concern — but it's worth naming.

Option 2: VPS or Cloud Instance

A small virtual machine from any cloud provider.

The case for it:

No hardware to manage. The instance runs regardless of what happens at your home. Easier to hand off or share if this eventually becomes a team system rather than a personal one.

The tradeoffs:

Cost. A small instance that can actually run this workload costs $15–40/month, ongoing. Over a year, that's the price of a decent Mac Mini.

Latency for local integrations. Anything that would benefit from local hardware access (filesystem, macOS APIs, low-latency local network) has to go over the network instead. This isn't a dealbreaker, but it does meaningfully limit the "local Mac capabilities" tier of what your assistant can do.

Security requires work. A cloud VM has a public IP by default. You need to harden it: firewall rules, SSH key-only auth, no root login, regular updates. Not impossible, but it's work you don't have to do on a local machine behind Tailscale.

Option 3: Container on Your Development Machine

The fastest way to start. `docker run` and you're running.

The problem: your assistant goes offline when your laptop sleeps. It goes offline when you travel. It goes offline when you close the lid.

For a system that delivers its value partly through proactive, always-on behavior — the morning briefing at 7am, the meeting prep that runs while you're asleep — this breaks the core promise. A container is a great development environment and a poor production environment.

The Recommendation

If you have a spare machine that can run always-on — a Mac Mini, a NUC, an old laptop — use it. The combination of low cost, native capabilities, strong security defaults, and superior developer experience is hard to beat.

If you don't, a small VPS is the right tradeoff. Accept the loss of native Mac capabilities, spend an afternoon hardening the instance, and you have something that works well.

Use a container on your dev machine to build and test. Don't run it as your production assistant.