Velociti Context Dispatch

Developer Guide

The Problem This Solves

Every product initiative begins with an idea/problem. The challenge has never been generating ideas, it's translating them into implementation-ready context that developers and AI coding agents can execute with confidence.

Traditionally, this translation happens through a series of meetings, Slack conversations, design reviews, and documentation. Product managers explain the same feature multiple times, developers fill in missing details with assumptions, and valuable context is lost with every handoff. The result is unnecessary rework, misaligned expectations, and products that don’t meet customer needs.

The **Context Dispatch** eliminates this translation gap.

Product teams define an Initiative in Velociti using customer insights, product strategy, requirements, and design context. The Context Dispatch then transforms that information into a comprehensive, implementation-ready artifact that includes functional requirements, user stories, UI behavior, workflows, technical considerations, data models, API expectations, and the surrounding product context needed to build the feature correctly.

Instead of interpreting fragmented requirements, developers receive complete product context in a format designed specifically for AI coding agents. Rather than repeatedly explaining the same feature, product managers create a single source of truth that aligns product, design, engineering, and AI coding tools around the same understanding.

The result is fewer assumptions, fewer clarification meetings, faster development, and significantly higher confidence that what gets built matches the original product intent.

What You Get

A Context Dispatch is a bundle of files generated per Initiative. Each one serves a specific purpose.

01 - Strategic Context

The "why." Who the user is, what problem they have, what success looks like. Read this once to understand what you're building and for whom. Don't skip it - it'll save you from building the technically correct thing in the wrong direction.

02 - Product Requirements

The "what." A story map of every user action broken down into steps and acceptance criteria. This is your checklist for what done looks like.

03 - UX Architecture

The "how it feels." Every screen defined with exact copy, component names, interaction logic, state management hints, and API calls per interaction. This is the most detailed file - treat it as your interaction spec.

04 - Data Schema

Models, fields, types, required vs optional, and API endpoint definitions with request and response shapes. Your source of truth for data structure.

05 - Frontend Scaffold

Recommended file structure, every component with its path, props interface, and description. You can follow this exactly or adapt it to fit your existing setup.

06 - Mock Data

Ready-to-use seed data that matches the schema. Drop it in as your dev fixtures so you can build and test without needing a real backend.

Optional Files (included when PM provides them)

07 - Design Context (optional)

Only present if the PM attached a design file when creating the initiative. This file translates the visual design into

written requirements - component layout, spacing, color usage, and any specific UI details extracted from the

design.

You will also find a design/ folder alongside this file containing the actual image the PM uploaded. Use both

together - the image as visual reference, the markdown file as the written spec derived from it.

Skills (Optional)

When a PM enables skills, the agent layers in extra context during generation based on those skills. So the files you get will have more targeted guidance baked in depending on what was turned on.

For example:

- **Security skill** - might add input validation notes, auth considerations, and data handling guidance
- **Architecture skill** - might add notes on component boundaries, separation of concerns, and scalability considerations
- **Technical Writing skill** - might improve the clarity and structure of specs and copy

You don't need to do anything differently as a developer. Just know that if you see extra context or guidance in the files, it came from the skills the PM had enabled.

Connecting via MCP

The Velociti MCP server lets you pull context Dispatch files directly inside your coding agent without downloading anything manually. You set it up once and then just ask for what you need in plain language.

What is MCP?

MCP (Model Context Protocol) is an open standard that lets coding agents connect to external tools and services. Think of it like giving your agent a direct line to VelocitiPM so it can fetch your context Dispatchs, check what initiatives are available, and know what you're working on - all without leaving your coding environment.

Setup (one time)

Pick the setup that matches your coding agent. You only need to do this once.

Claude Code

Run this in your terminal:

```
claude mcp add velociti --transport http https://mcp.velocitipm.com/mcp --scope user
```

Codex

Run this in your terminal:

```
codex mcp add velociti --url https://mcp.velocitipm.com/mcp
```

AGY (Google Antigravity CLI)

AGY uses a config file instead of a CLI command. Add this to your `~/.gemini/config/mcp_config.json`:

```
{
 "mcpServers": {
  "velociti": {
   "serverUrl": "https://mcp.velocitipm.com/mcp"
  }
 }
}
```

Then restart AGY and it will pick up the new server.

Other Coding Agents

Most coding agents that support MCP use a JSON config file. Look for an "mcpServers" section in your agent's settings and add:

```
"velociti": {
 "url": "https://mcp.velocitipm.com/mcp"
}
```

Check your agent's documentation for where to put the config file. The URL is always the same: `https://mcp.velocitipm.com/mcp`.

Authenticating

After adding the server, you need to authenticate once. The exact steps vary by agent but the flow is the same across all of them:

1. Try using a tool (e.g. "list my workspaces")
2. Your agent will detect it needs auth and prompt you
3. A browser window opens with the VelocitiPM login page
4. Sign in and you're done - your agent stores the token and handles refresh automatically

If your agent does not trigger the browser automatically, look for an "Authenticate" button in your agent's MCP settings panel.

Available Tools

Once connected, you can use these tools in plain language inside your coding agent.

get_active_session

Check your current workspace and initiative.

```
"What is my current workspace and initiative?"
```

list_workspaces

See all workspaces you have access to.

> "List my workspaces"
> "Show me all my projects"

set_active_workspace

Set which workspace you are working in.

> "Set Formula 1 as my active workspace"

list_initiatives

See what is available to work on. Only shows initiatives with completed context Dispatchs.

> "Show me open initiatives"
> "What's available to work on?"

set_active_initiative

Tell the system you are starting work on a specific initiative.

> "I'm working on the Newsletter Subscription initiative"

get_context_Dispatch_files

Pull all Context Dispatch files for your active initiative directly into your session.

> "Get context files for this initiative"
> "Show me the context Dispatch"
> "Pull the PRD"

Re-authenticating

If tools stop responding or you see auth errors, look for the MCP settings in your agent, find Velociti, and hit Authenticate again.

How to Use With Your Coding Agent

Step 1 - Get Your Context Dispatch Files

You have two options depending on how your PM shares the context Dispatch with you.

Option A - Via MCP (recommended)

If you have the MCP server set up (see above), just ask your agent directly:

> "Get context files for the onboarding initiative"

Your agent will pull all 6 files into the session automatically. No downloading, no attaching.

Option B - PM sends you the files

If your PM exports and sends you the files, you have a couple of options:

- **Attach them directly** - start a new session and attach all files
- **Add to your repo** - drop them in a folder like `context/` at the root of your project and reference the path when prompting

If you put them in your repo, just tell your agent where they are:

> "I have context Dispatch files in the /context folder.
> Read all of them before writing any code."

Step 2 - Let the Agent Read Everything First

Before any code gets written, make sure your agent has read all 6 files:

> "I have a context Dispatch for a new feature. Read all 6 files before
> writing any code. Start by confirming what we're building and
> what the file structure will look like."

Step 3 - Build the Scaffold First

Once the agent confirms it understands, create the file structure before any logic:

> "Start by creating the file structure from 05_Frontend_Scaffold.
> Create all files as empty shells with the correct component names
> and TypeScript interfaces. Don't implement anything yet."

Step 4 - Implement Screen by Screen

Work through one screen at a time using the UX Architecture file as your guide:

> "Now implement the OnboardingScreen. Use the interaction logic in
> 03_UX_Architecture for state management and API calls.
> Use the mock data from 06_Mock_Data_Seed as the data source for now."

Repeat for each screen.

Step 5 - Wire Up the Data Layer

Once screens are built:

> "Now implement the API functions in lib/api.ts based on the endpoint
> definitions in 04_Data_Schema. Keep them returning mock data for now."

Step 6 - Review Against Requirements

Before calling it done:

> "Cross-check the implementation against every acceptance criteria in

```
02_Product_Requirements. List anything missing or incomplete."
```

What the Context Dispatch Covers vs What You Still Own

The context Dispatch gets you to a working prototype fast. It does not get you to production.

What is covered

- Component structure and props
- State management patterns
- API contract shapes
- Data models
- UI copy and interaction logic
- Mock data for development

What you still need to do

- Connect to your real backend and database
- Add auth (the context Dispatch assumes you have it, it does not implement it)
- Adapt the file structure to your existing codebase conventions
- Replace mock data with real API calls
- Add error handling beyond the happy path
- Write tests
- Performance, accessibility, and security hardening

Think of it like a detailed architectural blueprint. The blueprint tells you exactly what to build and how it fits together. You still need to lay the actual foundation.

Tips

- **Read 01 first, always.** The strategic context will save you from going down the wrong path. Five minutes now saves hours later.
- **Don't fight the scaffold, adapt it.** If your codebase uses a different folder structure that is fine. The component names and interfaces are more important than the exact paths.
- **The UX Architecture file is your source of truth for behavior.** If you are unsure what should happen when a user clicks something, it is in there.
- **Mock data is your friend.** The seed file matches the schema exactly. Use it to build and test everything before your backend is ready.

- **Flag schema conflicts early.** If the data schema in file 04 conflicts with tables you already have in your DB, flag it before you build, not after.