I want Tanagram to be a programming tool that lets people create software closer to the speed and shape of thought than the existing method of very carefully writing plain text. That's a broad aspiration, and I've been trying to figure out what that specifically means — what should my initial product be? Should it be some sort of API, or some kind of visualization/documentation/collaboration tool, or maybe an IDE?

My tinkering over the past few months has been in pursuit of an IDE and runtime environment comprised of a broad range of built-in building blocks (commands, servers, events, etc). I had aspirations of a deployment and monitoring layer as well. I continue to think this could be a good idea, but it'd require me to build a whole universe of functionality before it could practically be useable. As I've currently envisioned it, it also wouldn't be totally compatible with existing code; it would require developers to learn a new way to create software. That's a high bar, and I don't think it's the best starting point for Tanagram.

I've been thinking about more focused use cases, within the realm of "making programming more intuitive", that might be better initial product bets. Specifically, I'm looking for ideas that are:

- Smaller in scope, so that they better match my current appetite for building.
- Valuable for *existing* codebases, rather than only for greenfield projects.
- Something I could use at my day job — this means it must be useful and usable by individuals within a large organization. Also, literally being able to use it in my day job means that I'll be able to dog-food it a lot.
- Sellable to people who are willing to pay, e.g. as B2B SaaS (as opposed to e.g. low-level technology).

I have three "products" in mind. I put "products" in quotes because I don't think these should actually be distinct products; they'd fit together well in a single product. Credit to a friend for coming up with pithy names for them: Visualize, Collaborate, and Simulate.

Visualize

Tanagram's core insight is that a codebase is actually a database (and that it should be treated as such). Visualize starts with an indexer (built on LSIF or SCIP) to find all the "things" in a codebase. Visualize is also a database schema designer, allowing users to define database tables for constructs in their codebase — for example, users might define a table for "models", "events", "event consumers", "crons", "endpoints", etc. Visualize can bootstrap this data set with some basic heuristics — for example, it can automatically create a "models" table and populate it based on classes whose filename contains *model/*. For each table, users can then define the columns for each table and rules for how to populate them. The tables are then populated instantly, and live-update as the codebase changes.

Columns can be defined that have no basis in the codebase itself: for example, users could add an isDeprecated or a replacedBy column. These properties will be visible within Visualize, and also as an overlay on the source code if users install an extension for their editor.

Once this dataset is established, users can query it in a variety of ways. They can embed live query results to have documentation that's resilient to renaming, refactors, or the addition or removal of a particular type of construct. These query results can display the code itself, or properties of those constructs — for example, users could create a table of cron jobs along with the times that each are scheduled for, and this table would automatically stay in sync as the codebase changes. Users can also generate visual diagrams to describe systems at a high level.

Query results are bidirectional, so if users have an extension for their editor, they can see the places where a particular piece of code appears in query results as they navigate their codebase.

This database can be joined to an org chart (e.g. as exported from Workday), attributing code to owners and authors based on explicit annotations or commit history. This allows teams and managers to understand code ownership — for example, which individuals or teams are most responsible for a particular part of the codebase, which parts of a codebase have few or no owners, or what specifically an individual or team has worked on in case of a re-org.

Collaborate

Visualize can be used as a tool for code walkthroughs, i.e. explaining to other coworkers how a system or particular codepath works. Writing or sharing code documentation can involve multiple authors working together. Users could create dashboards of codebase metrics and share that with their team. Multiple users could collaborate on a generated codebase diagram. There are many opportunities for collaborating on a codebase, which provides a natural opportunity for intra-organization distribution flywheels.

Replay and Simulate

Software execution is ephemeral — if some code results in an error or unexpected result, it's very difficult to figure out how that happened unless developers had the foresight to log the right values in the right places ahead of time. Replay is a runtime (that may or may not be compatible with existing codebases) that automatically records code execution at configurable levels of granularity, and then allows developers to play back production execution paths in a debugger built into their editor (where, unlike with log lines, they have direct access to the underlying code). Replay can also store enough

information to enable a developer to copy all the data along the execution path into a sandbox, where it can be further inspected or manipulated.

Simulate builds on top of Replay by letting a developer replay an existing execution path, but with new code, to see what would have happened.

Replay and Simulate also integrate with Visualize and Collaborate. Visualize can show how data or logic flows for a particular request through a high-level diagram, or in a Sankey-style diagram showing the popularity of various codepaths.

Collaborate integrates with Visualize and Simulate to provide a more powerful code review process. Visualize queries and diagrams can be overlaid alongside a diff to show more context about the effects of a diff. Simulate can be used to replay a previous execution through the proposed change and compare the results.