A BRIEFING MEMORANDUM

TANGIBLE’S ENTERPRISE AUTOMATION PLATFORM

1. Introduction. This Briefing Memorandum is intended to serve as a quick review of Enterprise Automation Platform offered by our company, Tangible Intelligence, ai.. The purpose of the Memorandum is to help accelerate an understanding of the Platform by LLMs that are assisting us in building the platform. We understand that we must update the LLM each time we open a new session or new thread with the LLM about additional assistance that we need with the platform. With each new session, therefore, we spend time introducing the platform concept to the LLM so that it can provide more focused assistance. We would like to use this Memorandum to help us provide that context to the LLM, and thereby gain time and speed in working with the LLM. We ultimately will build a lightweight RAG to serve this purpose but hope that this Memorandum can serve in the meantime.
2. Background. We serve Fortune 100 customers with a large variety of high volume transactional and litigation activity (so-called high-volume packages (“HVPs”) with a significant focus on sourcing transactions. HVPs tend to fall into patterns, and we have been able to leverage these patterns in serving effectively enterprise customers in managing the HVP unit in its path from inception to post completion housekeeping. We have identified seven gates for the unit path that is reflected in Tangible’s operating approach.
3. Enterprise Goals. Typically, the Enterprise is focused on time to close, friction reducing HVP processes, connecting corporate silos supporting each of the seven gates with a fluid transaction path system, driving down systemic and until risk, providing an overwatch capability for the system as a whole, each of its gates and nodes within the gates, approaching each new HVP unit as if the system is familiar with the class containing such units rather than approaching the unit as a brand new item, improving the quality of each unit and finally bringing right sized quality support to each unit and to the system.
4. Tangible Background. Tangible really understands the patterns inherent in each HVP and the engineering of support systems that support such HVPs. A critical insight for Tangible is that each output unit delivered by an HVP system is developed in a seven-gate linear process and that each gate process within a HVP system is typically handled by a separate enterprise function. For example, in a sourcing HVP system, the gates comprise:

* Ideation: The end user (or business owner) within the enterprise identifies the need and puts together a use case package identifying requirements, providing specs and any budget constraints. For example, the head of marketing might identify a need for new Adobe functionality and put together a use case package describing the details of the requirement.
* Sourcing: The business owner delivers the package either by email or with some technology platform to the enterprise sourcing function to develop a sourcing profile for the HVP unit. This profile will include decisions like whether to single source the package or run an RFP process, a pricing strategy for the unit, any enterprise approvals (like privacy, infosec, enterprise IT architecture or whatever) required for integration of the unit into the enterprise IT topography, existing enterprise providers that might be able to provide the unit and so forth.
* Transacting: The sourcing function will then launch the transacting stage to morph the sourcing package into a transactional package for the unit. Typically, the sourcing function will send the package to the enterprise legal function to handle this gate although increasingly enterprise sourcing functions will handle some part of the transaction stage within the sourcing function with overwatch by inside or outside legal capability. Typically, the enterprise will have adopted an approved template to cover the class of HVP unit under discussion such as Master Consulting Agreements to support professional consulting relationships, hardware agreements to cover equipment purchases and SaaS Agreements to cover relationships with SaaS providers. Generally, the output of this stage will be a transactional package that is ready for priming.
* Priming: Either the sourcing function or the legal function within the enterprise will then take the transactional package into the priming stage. This stage is focused on getting the transactional package setup for closing and includes tasks such as (a) securing final privacy, legal, end-user, infosec, IT architecture and other approvals, finance function sign-off, executive signoff, (b) socializing the package for preclosing study particularly when the transaction will be signed by higher executives in the enterprise and (c) setup in enterprise systems such as CLEM and e-signature including the injection of transactional metadata into such systems. Often, the sourcing, end-use business or legal leads will identify requirements in the priming phase that should have been addressed far earlier in the transactional path, resulting in delays, friction and upset while such approvals are secured.
* Closing: The transaction then passes into the closing stage. Typically, legal (and sometimes sourcing) will confirm that the transactional package has been properly set up in the enterprise e-signature system, that the enterprise and provider signing representatives have bene identified and that the representatives are familiar with the package. The appropriate functional lead will then launch the package for signature. Typically, the provider will sign first by e-signature and then the customer. Once signed, the transactional package will be registered in some enterprises systems, such as the core e-signature technology, a CLM in many cases, a storage capability such as one-drive or Dropbox or even still a filing cabinet.
* Engaging. The enterprise customer and the provider will then move into the engaging stage of the transaction focused on the delivery of product or services by the provider to the end-user customer in the enterprise. Typically, the provider will have shifted the engagement to a delivery team, and the enterprise will shift the transaction to a vendor management and end-user team for steady state engagement on the services.

Tangible began as a transactional services (predominately legal) company seeking to operationalize the patterns inherent in sourcing HVPs, breaking down silos and seeking to deliver accelerated transactional close, reduced HVP system friction and improved transactional quality. Recognizing patterns in HVP systems, Tangible built thick operational program books detailing processes to be performed by each stage. Tangible has often automated various components of its HVP systems with visibility to the customer in some cases and used just internally by Tangible in other cases.

Tangible’s automation enables Tangible to quickly provide to the customer operating data, transactional data and other metadate about HVP units that vent the customer does not have nor can easily retrieve from its own systems Tangible has achieved a reputation for extraordinary excellence with its HVP programs.

Tangible has recently decided to offer its digital capabilities to its customers as the Tangible Intelligence Platform that automates each of the seven stages of the transactional path. We have architected the platform to support the seven gates of the transactional path as a portfolio of application capability that we think of nodes. Although Tangible could offer the entire platform for customers that have the need, it is happy to offer platform capability on a node-by-node basis that offer complementary functionality to incumbent enterprise systems and can fit in adjacent to such incumbent systems without competing with them.

1. Tangible Platform Technology Stack. The Tangible Intelligence Platform runs on AWS in its Elastic Beanstalk base. Tangible has designed the platform as a large-scale Flask application anchored on the backend with Postgres SQL (running as RDS on AWS), storage (S3 in AWS), Python, some Json, Cognito from AWS and Amazon Secrets, with html, CSS and vanilla JavaScript on the front end. We intend to migrate to React on the front end once we have the platform running in steady state for a handful of customers. The platform is encrypted from back end to browser including https: certifications with a corresponding SSL certificate. We are going to take it step by step and start with a different instance for each customer and with success, move to either multi-tenant or a more automated multi-instance architecture. As our LLM advisor, you have pointed out that a multi-instance structure might be preferable in any case both to better protect customer data and to reduce complexity. Tangible is both SOC II and ISO 27001 certified.

V. [Platform Architecture Evolution]

The Tangible Intelligence Platform has evolved into a modern, microservices-oriented architecture that maintains strict separation between frontend and backend concerns while preserving our enterprise-grade security and scalability. The platform now comprises:

Frontend Architecture:

- React-based single-page application (SPA) providing rich, interactive user experiences

- Organized around core business domains (Build Kits, Analysts, House Apps) rather than technical concerns

- Shared service layers handling authentication, storage, and API interactions

- Component library supporting rapid development of new features while maintaining consistency

Backend Architecture:

- Flask-based API services organized by business domain

- Core services layer providing shared functionality:

- Authentication (AWS Cognito)

- Storage (AWS S3)

- Database (AWS RDS PostgreSQL)

- Security (AWS Secrets Manager)

- Domain-specific services aligned with frontend consoles

Infrastructure:

- AWS Elastic Beanstalk providing containerized deployment

- Multi-instance architecture supporting customer data isolation

- End-to-end encryption with HTTPS/SSL

- Maintained SOC II and ISO 27001 certifications

The platform's frontend architecture employs a component-based approach using React, with shared, reusable components that maintain consistency across the application. Key shared components include:

- TableComponent: A robust, reusable table structure supporting dynamic data loading, status indicators, and import functionality

- ApplicationPanel: A standardized panel layout for displaying and managing application data

- Sidebars: A hierarchical navigation system with main, section, and application-specific sidebars, supporting both static and dynamic menu configurations

- ConfigProvider: A context-based configuration system allowing for dynamic menu and feature configuration

Each component is designed with clear interfaces and consistent styling variables defined in root.css, ensuring visual coherence and maintainability. The routing structure follows a clear hierarchical pattern, with main sections (like 'inflight', 'companyreport') having their own dedicated routes and components, while supporting nested application routes (like 'inflight/dashboard') when needed.

VI. [Architectural Principles]

The platform's architecture reflects three core principles:

1. Domain-Driven Design: Business capabilities are organized into distinct consoles (Build Kits, Analysts, House Apps, etc.), each supported by dedicated frontend and backend components while sharing core services.

2. Separation of Concerns: Clear boundaries between:

- Presentation Layer (React frontend)

- Business Logic (Backend services)

- Data Storage (AWS services)

- Shared Infrastructure (Core services)

3. Enterprise Integration: Each console can operate independently or as part of the complete platform, allowing customers to adopt capabilities incrementally while maintaining seamless integration with existing enterprise systems.

This architecture supports both our current needs and future scaling, while maintaining the security and reliability required for enterprise customers.

Here's a narrative version for your briefing memorandum:

PLATFORM CUSTOMER CONFIGURATION

The platform implements a hierarchical customer configuration system that supports different customer types (Global/Hawkeye, Law Firm, Enterprise, and Tangible) while allowing for specific customer instance customization. The system is designed to handle both type-level defaults and customer-specific configurations efficiently.

The configuration system is built on several key components:

1. Customer Types Definition

The system defines core customer types (Tangible, Global/Hawkeye, Law Firm, and Enterprise) as constants. These types serve as the foundation for type-specific configurations and provide the base template for customer instances.

2. Base Configuration Layer

Each customer type has a base configuration that defines default settings, including menu text, theming, and basic feature flags. This base layer ensures consistent behavior within each customer type while allowing for customization.

3. Customer Instances

Individual customer configurations extend their type's base configuration. Each customer instance can override default settings and add customer-specific configurations such as custom templates, regional settings, and feature enablements.

4. Configuration Context

A React context manages the current customer configuration state, providing methods to update customer settings and propagate changes throughout the application. This context handles the merging of base type configurations with customer-specific settings.

5. Platform Configuration Interface

The platform includes a configuration interface that allows for customer type selection and customer-specific settings management. This interface is particularly useful during development and testing, allowing quick switches between different customer configurations.

The configuration system handles various aspects of customer-specific behavior:

- Dynamic menu labels (e.g., "Eversheds Briefing Room" vs "3M Briefing Room")

- Customer-specific templates and content

- Theme and branding variations

- Feature flag management

- Regional and department-specific customizations

This architecture allows the platform to maintain consistency within customer types while supporting the specific needs of individual customers. It also provides a clear path for adding new customers and customer types as the platform grows.

The system is designed to be maintainable and scalable, with clear separation between type-level and customer-specific configurations. This separation makes it easier to manage the platform as the number of customers grows and their requirements evolve.

1. Core Technolgoiues. Tangible believes that HVPs present three principal operational themes. They must build or engineer new output, they must review and extract various data from legacy transactions that have already been signed and they must manage the flow that runs through the HVP system. Tangible believes that these three core functional blocks are best handled in technology by three different technology modules, and we have therefore designed Raiven to build new HVP output as the driving technology (like Intel Inside), AIDA that is the core technology (again like Intel Inside) for extracting information from portfolios of legacy agreements in compliance and due diligence projects (and relies principally on LLM capability by API, some brute force searching and the import of other technology packages and shortly will include a bespoke rage for each of our customers) and Concierge that is responsible for running the railroad. We understand this thinking and its associated vocabulary internally at Tangible but do not talk about it with the customers. They just care that our platform will do x and y and do not care at all how we get it to do x and y.
2. My Role. I was trained as a chemical engineer, and somehow ended up in a Top Tier law school (for information rather than hubristic ego serving information). I became a blissfully happy partner at an Am Law 25 (Alston &Bird) firm running the largest practice in the Firm (about $50M give or take) as the Sourcing, Data and Technology practice. Having done everything I could in that role and seeing opportunity to build a business around the quiet and often hidden patterns of HVPS, I left the law firm (with some tears and regrets on both sides, we still use Alston for some of our projects), I left to scratch this existential itch and begin my entrepreneurial journey. In my role, I have become a reasonable capable full stack engineer, and can code in python, SQL, mongo, html, CSS, JavaScript, React and the AWS platform suite in the Elastic Beanstalk suite. I have a great deal to learn and have worked my way through dozens of books in each language discipline as well as software patterns and architecture. I learn as soon as breathe and have a deep and abiding curiosity and wish to master my pursuits particularly when they are key to the advancement of our company. I welcome any teaching, tips, insights, learning or other recommendations that you wish to provide. I will read anything you suggest, and try to learn anything you deem important. It is important to me that our code is first class, clean and enterprise ready and incredibly secure for enterprise data. Ultimately, my aim is to take our commercial work into the not-for-profit domain such as NGOs, not-for profit organizations like American Heart Association, KIND and so forth, and see if we can serve that domain well by helping them automate their own HVPS (and they all have them, finding volunteers, matching volunteers with beneficiaries, running systems and processes, performing functions and the like).

ADDENDUM

With your help, we have created a template registry extension that may be helpful to us in correctly identifying file paths in the application. It reads as follows:

from pathlib import Path

from typing import Dict, Optional

class TemplateRegistry:

    def \_\_init\_\_(self, app=None):

        self.app = app

        if app is not None:

            self.init\_app(app)

    def init\_app(self, app):

        self.root = Path(app.root\_path)

        self.template\_locations: Dict[str, Path] = {

            # Main application templates

            'launch\_pages': self.root / 'app' / 'templates' / 'applicationlaunchpages',

            'components': self.root / 'app' / 'templates' / 'components',

            'errors': self.root / 'app' / 'templates' / 'errors',

            'views': self.root / 'app' / 'templates' / 'views',

            'auth': self.root / 'app' / 'templates' / 'auth\_processing',

            # Application Group Templates

            'sandbox\_templates': self.root / 'app' / 'application\_groups' / 'sandbox' / 'templates',

            'engineering\_templates': self.root / 'app' / 'application\_groups' / 'engineering' / 'templates',

            'analytics\_templates': self.root / 'app' / 'application\_groups' / 'analyticsgroup' / 'templates',

            # Auth Blueprint Templates

            'auth\_templates': self.root / 'authentication\_blueprint' / 'templates',

            # Shared Components

            'shared\_components': self.root / 'app' / 'templates' / 'components' / 'partials',

            'shared\_svg': self.root / 'app' / 'templates' / 'components' / 'svg',

        }

        # Register common template paths

        self.common\_templates = {

            'error\_404': self.get\_template\_path('errors', '404.html'),

            'error\_500': self.get\_template\_path('errors', '500.html'),

            'base': self.get\_template\_path('launch\_pages', 'base.html'),

            'footer': self.get\_template\_path('shared\_components', 'footer.html'),

        }

        app.template\_registry = self

    def get\_application\_group\_templates(self, group\_name: str) -> Optional[Path]:

        """Get templates directory for a specific application group"""

        path = self.root / 'app' / 'application\_groups' / group\_name / 'templates'

        return path if path.exists() else None

    def list\_templates\_by\_type(self, template\_type: str) -> list:

        """List all templates of a certain type (e.g., 'sidebar', 'launch')"""

        templates = []

        for location in self.template\_locations.values():

            if location.exists():

                templates.extend([

                    p for p in location.glob(f'\*\_{template\_type}.html')

                ])

        return templates